UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Rapports Techniques

N°136

Programme 2
Calcul symbolique, Programmation et Génie logiciel

L'ARITHMÉTIQUE GÉNÉRIQUE DE SISYPHE

José GRIMM

Décembre 1991

Institut National
de Recherche
en Informatique
et en Automatique

Sophia Antipolis
B.P. 109
06561 Valbonne Cedex
France
Tél.: 93 65 77 77

Programme 2

L'arithmétique générique de Sisyphe

The generic arithmetic of Sisyphe

José Grimm

Projet SAFIR

INRIA, Sophia Antipolis
2004, Route des Lucioles
06561 Valbonne Cedex

email : grimm@sophia.inria.fr

L'arithmétique générique de Sisyphe

Résumé

Ce rapport décrit l'implémentation de l'arithmétique dans le système de calcul formel Sisyphe. Elle est basée sur l'existence dans LeLisp de primitives de bas niveau (par exemple la multiplication de deux entiers 16bits avec résulta sur 32 bits), et d'une arithmétique générique (destinée à être étendue).

Dans une première partie nous utiliserons les primitives pour implémenter une arithmétique sur les entiers positifs de taille arbitraire. Dans une seconde partie, nous utiliserons l'arithmétique générique pour définir les entiers signés et les fractions rationnelles. Finalement nous introduirons les nombres complexes et les fonctions transcendantes et décrirons les fonctions d'entrées-sorties.

Abstract

This report describes the implementation of the arithmetics of the computer algebra system Sisyphe, based on the existence in LeLisp of low level primitives (such as multiplication of two 16bit integers yielding a 32bit result) and a generic arithmetic (meant to be extended).

In a first part, we use the primitives to implement undigned bignums of arbitrary size. In a second part, we extend it to signed integers and rational numbers, using the generic arithmetic. Finally, in the last part complex numbers and transcendental functions are defined and I/O functions are described.

Chapitre 1

Introduction

Ce rapport décrit l'implémentation de l'arithmétique rationnelle de SISYPHE faite en LELISP version 15.24. Dans une première partie, nous décrirons les algorithmes manipulant les nombres entiers en précision arbitraire. Dans les chapitres suivants, nous décrirons l'arithmétique générique et certains algorithmes particuliers.

1.1 Historique

Ce logiciel a été écrit en 1987 pour servir de base pour un système de calcul formel, qui a reçu plus tard (en 1988) le nom de SISYPHE. Le système SISYPHE est actuellement développé par le projet SAFIR, projet commun à l'INRIA, au CNRS et à l'Université de Nice-Sophia Antipolis.

La première implémentation de ce logiciel a été faite dans la version 15 de LELISP, elle était entièrement écrite en LLM3, et utilisait des optimisations spécifiques sur Multics. A l'époque, il y avait dans LELISP une seule implémentation de l'arithmétique générique (qui existe encore, et décrite dans le manuel de référence de LELISP [2, chapitre 10]). Cette représentation utilise des arbres équilibrés pour représenter des entiers (il s'agit de cellules de listes étiquettées ou *tcons*), et a comme principal avantage d'être extrêmenent efficace sur les entiers de taille intermédiaire (quelques mots machines), et assez efficace pour les fonctions comme l'addition et la comparaison. Par contre, elle est assez lente pour les calculs de pgcd. Or, il est arrivé, lors de la factorisation de polynômes, de calculer un pgcd sur des entiers ayant plusieurs milliers de chiffres. Dans ce cas cette implémentation donne des résultats catastrophiques : plusieurs *milliers* de fois plus lent que notre arithmétique [Voir la table section 2.6].

Il devint alors évident qu'il fallait adopter une représentation des entiers sous forme de suite de bits (vecteur de chiffres ou chaînes de caractères). La question qui se posait était de savoir s'il fallait ou non étendre les primitives Lisp. Comme cela n'a pas été fait, il restait deux solutions.

La première solution consistait à prendre les primitives actuelle de Lisp. C'est l'option choisie ici, le code LLM3 a alors été traduit en Lisp, sans perte d'efficacité notable (ceci est essentiellement dû au rajout dans LELISP d'un nouveau compilateur, "complice", nettement plus efficace que le compilateur par défaut).

La seconde stratégie consistait à écrire de nouvelles primitives, plus puissantes (par exemple, une fonction qui additionne deux vecteurs de chiffres) en assembleur, ou plus généralement dans un langage comme C. Cette stratégie a été utilisée dans [1]. L'ensemble de ces primitives recouvre

à peu près le chapitre 2, et doit être complété par des fonctions Lisp, telles que celles décrites dans les chapitres suivants.

1.2 L'avenir

Notre logiciel est bien adapté à la version 15 de LeLisp. L'un des problèmes qui va se poser dans l'avenir concerne le portage en version 16. Cette nouvelle version possède des traits caractéristiques qui sont soit faciles à intégrer (liaison lexicale, fermetures), soit nécessitant une modification du code, (par exemple, nouvelles structures, etc.), soit nécessitant une modification des algorithmes.

C'est essentiellement pour résoudre ce dernier point que nous avons décrit en détails tous les algorithmes. Deux points nous semblent cruciaux : le premier est que l'arithmétique sur les petits entiers passe de 16 bits à 30 bits. Le second est que, pour l'instant, aucune des primitives utilisée ici et décrites dans la section suivante ne sont implémentées dans la version 16. A nouveau se pose la question : faut-il rajouter comme primitives, les anciennes, ou coder de nouvelles primitives en C?

1.3 Les primitives Lisp à notre disposition

Les algorithmes que nous allons présenter dans le premier chapitre sont pour la plupart décrits dans [3, section 4.3] et adaptés à Sisyphe. On suppose savoir manipuler des entiers en base $E = e^p$. Dans la version courante de Sisyphe, on suppose $e = 2$ et $p = 16$. En principe ces valeurs particulières ne devraient pas avoir d'influence sur les algorithmes, mais ceci n'est pas toujours exact. C'est pour cela que nous préciserons chaque fois que cela est nécessaire l'influence de l'exposant (en ce qui concerne la base e, on suppose qu'elle est toujours égale à 2).

Précisons ce qu'on entend par savoir manipuler des entiers en base E. D'une part, on utilise une variable globale `C` (pour "carry", c'est la retenue courante, en interne, elle s'appelle `#:ex:regret`). Le fait d'utiliser une variable globale est important dans la mesure où une fonction ne peut rendre qu'une seule valeur. On préfère modifier `C` et rendre une valeur plutôt que de rendre une liste (les fonctions utilisées ici sont des fonctions de base, on ne peut se permettre d'allouer de la mémoire de façon injustifiée).

On dispose des cinq fonctions suivantes. D'une part, la fonction `ex+` prend deux arguments a et b; elle calcule la somme $a + b +$ `C` sous la forme $CE + d$. La quantité C est mise dans `C`, le résultat de la fonction est d. Notons que $a + b +$ `C` $\leq 3E - 3 < E^2$ pourvu que $E \geq 2$. La fonction `ex*` prend trois arguments a, b et c. Elle calcule $ab + c +$ `C` sous la forme $xE + y$. Elle rend y, et positionne x dans `C`. Notons que $ab + c +$ `C` $\leq (E-1)^2 + 2(E-1) = E^2 - 1$.

La fonction `ex/` prend deux arguments a et b. Elle réalise la division euclidienne `C`$E+a = xb+y$. Elle positionne y dans `C` et rend x. On suppose `C` $< b$ (en particulier $b > 0$), d'où $x < E$. Comme c'est une fonction de bas niveau, aucun test n'est fait, n'importe quoi peut se produire si cette condition n'est pas satisfaite.

La fonction `ex-` prend un argument a, elle calcule $E - 1 - a$. Définissons la fonction `ex--`(a, b) par `ex+`$(a$,`ex-`$(b))$. Elle calcule donc $E + a - b +$ `C` $- 1$. Dans le cas `C` $= 1$, si $a \geq b$, elle rend donc $a - b$ et laisse `C` à 1. La quantité `C` est appelée retenue dans le cas de l'addition et de la multiplication. Dans le cas de la soustraction, la quantité $1 -$ `C` est appelée emprunt. Finalement on dispose d'une fonction `ex?` qui compare deux arguments a et b. Elle rend 0 si $a = b$, 1 si $a > b$ et $E - 1$ si $a < b$.

Chapitre 2

Arithmétique en précision arbitraire

2.1 Introduction

Les algorithmes décrits ici manipulent des entiers positifs, appelés nombres internes. Il est facile de créer plusieurs types de nombres à partir de ceux-ci. Par exemple, dans SISYPHE, un entier est la donnée d'un signe et d'un nombre interne, un rationnel est la donnée d'un signe, de deux nombres internes (numérateur et dénominateur) et d'un indicateur disant si numérateur et dénominateurs sont premiers entre eux. Les entiers seront expliqués dans le prochain chapitre. La représentation interne des polynômes de SISYPHE utilise également des entiers internes, mais non l'arithmétique générique.

Un entier entre 0 et $E-1$ sera appelé un chiffre. Un nombre interne sera soit un chiffre, soit un nombre normalisé. Par nombre normalisé on entend un tableau de chiffres de la forme $A = [a_n, \ldots, a_0]$ où a_n est non nul, $n > 0$, qui correspond au nombre $\sum a_i E^i$. Un nombre non normalisé est alors un nombre $a = \sum a_i E^i$ qui se trouve dans le tableau A, lequel tableau est de la forme $[b_0, \ldots, b_k, a_n, \ldots, a_0, c_0, \ldots, c_q]$. On dira dans ce cas que a_n est le premier chiffre utile et a_0 est le dernier chiffre utile. Ces deux chiffres sont repérés par deux indices i_A et t_A (i_A est l'indice du premier chiffre utile et t_A est l'indice du dernier chiffre utile). On utilise huit variables globales pour repérer les indices utiles, à savoir i_A, i_B, i_C, i_D, i_E, t_A, $t_B$et t_C (dans le code Lisp, ces variables s'appellent `:indiceA` ou `:tailleA`, etc). Les indices commencent à 0, de telle sorte que si a occupe le vecteur A en entier, donc si $A = [a_n, \ldots, a_0]$, on a $i_A = 0$ et $t_A = n$, et le vecteur A est de taille $n+1$. Nous noterons $t(A)$ la taille d'un vecteur A. Les fonctions externes ne font aucune hypothèse sur les variables i_A, etc, ce sont uniquement les fonctions internes qui supposent que certaines conditions sont satisfaites.

Nous supposerons savoir manipuler les indices. En particulier, on sait incrémenter et décrémenter un indice sans utiliser la variable `C`. L'implémentation dans SISYPHE est la suivante : un indice est un chiffre. Si $i < E/2$ on considère que i est positif, et si $i \geq E/2$ on considère que i est négatif (valant $i - E$). Toutes les opérations sur les indices se font modulo E (en particulier on identifiera $E-1$ et -1). La taille d'un vecteur est au plus $E/2 - 1$ (donc est positive avec nos conventions). On pourra donc écrire "tant que $i \geq 0$ faire ..., décrémenter i" sans se poser de problèmes métaphysiques. Si on alloue un vecteur, on suppose que ce vecteur ne contient que des zéros.

Notons que les seuls tests d'overflow sont réalisés lors de l'allocation d'un vecteur : si la taille est négative ($\geq E/2$) une erreur est déclenchée. On se gardera donc de phrases du genre "allouer un vecteur de taille $3n$", par contre on autorise "allouer un vecteur de taille $n+m$".

Les algorithmes que nous allons présenter ici sont utilisés effectivement dans SISYPHE, avec les mêmes noms, modulo quelques optimisations (le cas typique concerne `nunderflow` qui est formé de quatre fonctions, pour des raisons de pure efficacité). Toutes les fonctions internes commencent par la lettre "i", certaines fonctions auxilliaires commencent par la lettre "a". Toutes les fonctions externes commencent par la lettre "n". Elles prennent en argument des nombres internes normalisés (à part `nunderflow`) et rendent des nombres normalisés. Contrairement aux fonctions internes, les fonctions externes ne modifient jamais leurs arguments. Ces fonctions ne sont pas accessibles directement par l'utilisateur. Elles sont dans le package **#:ed:r** (non indiqué pour rendre les algorithmes plus lisible). Les fonctions utilisateur sont décrites dans les chapitres suivants.

On aura besoin dans la suite des deux fonctions suivantes, qui sont faciles à implémenter. Supposons que $A = [b_0, \ldots, b_k, a_n, \ldots, a_0, c_0, \ldots, c_q]$. La fonction `afind0-beg` appliquée à A et aux indices i_A et t_A rend le plus petit i tel que a_i est non nul et a_j est nul pour $j > i$. La fonction `afind0-end` appliquée à A et aux indices i_A et t_A rend le plus grand i tel que a_i est non nul et a_j est nul pour $j < i$. Finalement, le facteur de normalisation d'un vecteur A est le facteur de normalisation de son premier chiffre a_n. C'est le plus grand chiffre e^k tel que $a_n e^k < E$, donc $E/e \leq a_n e^k < E$. On suppose qu'il existe une fonction de décalage qui permette de faire ce calcul sans modifier la variable `C` (dans le cas contraire, on peut utiliser `ex*`, en sauvant `C` dans une variable locale). De la même manière, il est possible de trouver k tel que $e^k \leq a_n < e^{k+1}$. Cet entier k est appelé le nombre de bits du chiffre a_n. Le nombre de bits du nombre a est donc $k + np$ (pourvu que a_n soit non nul).

2.2 Addition et Soustraction

Le code de l'addition et de la soustraction est relativement simple. Nous ne le commenterons donc pas.

Algorithme 1. (Addition des entiers) *Les variables locales et paramètres utilisés sont x et y des entiers internes, X et Y des vecteurs de chiffres, a un chiffre et i, j, n et s, des indices.*

Procédure nadd, addition externe. *Paramètres x et y.* [Cette fonction rend la somme des deux nombres internes sous forme d'un nombre interne].

Si x et y sont des chiffres, poser `C` $= 0$, $a =$ `ex+`(x, y). *Si* `C` $= 0$, *rendre a, sinon le vecteur* $[$`C`$, a]$.

Si seul x est un chiffre, rendre `iadd3`(x, y).

Si seul y est un chiffre, rendre `iadd3`(y, x).

Sinon [aucun des deux n'est un chiffre], *si x est plus long que y, copier x, et appeler* `iadd1` *sur x, y et $t(x)$, et dans le cas contraire faire la même chose, en échangeant x et y.*

Procédure iadd1. *Paramètres X, Y et n* [n taille de X, au moins celle de Y].

Poser `C` $= 0$, $i = n - 1$, $j = t(Y) - 1$.

Tant que $j \geq 0$, remplacer X_i par `ex+`(X_i, Y_j) *et décrémenter i et j.*

Rendre `iadd2`(X, i, n).

Procédure iadd2. *Paramètres X, i et n* [n taille de X, propager la retenue à partir de i].

Tant que $i \geq 0$ et `C` *non nul, remplacer X_i par* `ex+`$(X_i, 0)$ *et décrémenter i.*

Rendre `noverflow`(X, n).

Procédure iadd3. *Paramètres a et Y.*

Poser `C` $= a$, *et copier Y.*

Soit s la taille de Y. Rendre `iadd2`$(Y, s-1, s)$.

Procédure noverflow. *Données X et n* [Mettre la retenue avant le premier chiffre de X, n est la taille de X, rendre un nombre normalisé].

Si `C` $= 0$ *et $n = 1$, rendre X_0.*

Si `C` $= 0$, *rendre X.*

Sinon, allouer un vecteur de taille $n+1$, y copier X cadré à droite. Positionner `C` *en position 0. Rendre ce vecteur.*

On va maintenant définir la soustraction. On suppose qu'elle a un sens, donc que $x \geq y$. En particulier la taille de x est au moins la taille de y.

Algorithme 2. (Soustraction des entiers) *Les variables et paramètres utilisés sont x et y des entiers internes, X et Y des vecteurs de chiffres, i, j, et k, des indices.*

Procédure ndiff, soustraction externe. *Paramètres x et y* [rend la différence $x - y$].

Poser `C` $= 1$.

Si x est un chiffre, rendre `ex--`(x, y).

Si y est un chiffre, soit $i+1$ la taille de x. Remplacer x_i par `ex--`(x_i, y). *Appeler* `idiff2` *sur x et $i-1$.*

Sinon, copier x, rendre `idiff1`$(x, y, t(x)-1)$.

Procédure idiff1. *Paramètres X, Y et i* [$i+1$ est la taille de X, suppose `C` $= 1$].

Soit $j+1$ la taille de Y.

Tant que $j \geq 0$, remplacer X_i par `ex--`(X_i, Y_j), *décrémenter i et j.*

Rendre `idiff2`(X, i).

Procédure idiff2. *Paramètres X et i* [propager l'emprunt à partir de i].

Tant que $i \geq 0$ et `C` $= 0$, *remplacer X_i par* `ex+`$(X_i, -1)$, *et décrémenter i.*

Rendre `nunderflow`$(X, 0, t(X)-1)$.

Procédure nunderflow. *Données X, i et j* [rendre un entier normalisé extrait de X entre les indices i et j].

Soit $k =$ `afind0-beg`(x, i, j).

Si $k > j$ rendre 0.

Si $k = j$, rendre X_k.

Si $k = 0$, et X est de taille $j+1$, rendre X.

Dans les autres cas, rendre la partie de X entre les indices k et j, en la copiant dans un nouveau vecteur.

2.3 Multiplication

Nous allons proposer ici l'algorithme classique de multiplication. Il est un peu long car on teste plusieurs cas particuliers, et on optimise les multiplications par les puissances de E, par 0, par 1, etc.

Algorithme 3. (Multiplication des entiers) *Les paramètres et variables locales utilisés sont x et y, des nombres internes, X, Y et Z, des vecteurs de chiffres, a, b et c des chiffres, et i, j, k, k_0, t_1, t_2, s_1 et s_2, des indices.*

Procédure ntimes, multiplication externe. *Paramètres x et y* [rend le produit xy].

Si x et y sont des chiffres, rendre `ntimesfix`$(x, y, 0)$.

Si x est un chiffre, rendre `ntimesfix1`(x, y).

Si y est un chiffre, rendre `ntimesfix1`(y, x).

Sinon, soient $t_1 + 1$ et $t_2 + 1$ les tailles de x et y. De plus soit s_1 (resp. s_2) le résultat de `afind0-end` *appliquée à x (resp. y) en entier.*

Si $s_1 = s_2 = 0$, rendre `ntimesfix`$(x_0, y_0, t_1 + t_2)$.

Si $s_1 = 0$, poser $t_A = t_2 + 1$ et $t_B = s_2$, et rendre `ntimesc1`(y, x_0, t_1).

Si $s_2 = 0$, poser $t_A = t_1 + 1$ et $t_B = s_1$, et rendre `ntimesc1`(x, y_0, t_2).

Poser sinon $i_A = 0$, $i_B = 0$, $t_A = s_2$, $t_B = s_1$, $t_C = s_1 + s_2$, allouer un vecteur Z de taille $t_1 + t_2 + 1$. Appeler `itimes0` *sur x, y et Z, puis* `noverflow` *sur le résultat et $t_1 + t_2 + 1$.*

Procédure ntimesfix. *Paramètres a, b et i* [on calcule abE^i].

Poser `C` $= 0$, $c =$ `ex*`$(a, b, 0)$.

Si `C` $\neq 0$, *allouer un vecteur de taille $i + 2$. Y mettre* `C` *en position 0, et c en position 1, et rendre ce vecteur.*

Si $i = 0$, rendre c.

Sinon, allouer un vecteur de taille $i + 1$, y mettre c en position 0, et le rendre.

Procédure ntimesfix1. *Paramètres a et Y.*

Si $a = 0$, rendre 0.

Si $a = 1$, rendre Y.

Sinon rendre `ntimesfix11`(a, Y).

Procédure ntimesfix11. *Données a et Y.*

Positionner t_A à la taille de Y, et t_B au résultat de `afind0-end` *appliqué à Y en entier.*

Rendre `ntimesc1`$(Y, a, 0)$.

Procédure ntimesc1. *Paramètres X, a et i* [Calculer aXE^i; X est de taille t_A, normalisé, il y a des 0 après t_B].

Si $a = 1$, et $i = 0$, rendre X.

Sinon, copier X cadré à gauche dans un vecteur de taille $i + t_A$.

Si $a = 1$, rendre X.

Sinon, positionner i_B à 0, appeler `itimesc0` *sur X, a et 0. Appeler* `noverflow` *sur le résultat.*

Procédure itimesc0. *Paramètres X, un vecteur de chiffres (partie utile entre i_B et t_B), a et b* [Calculer $Xa + b$. En cas d'overflow, essaie de décrémenter i_B pour mettre l'overflow dans le vecteur X. Sinon l'overflow sera dans `C`].

Poser `C` $= b$, $i = i_B$ *et* $j = t_B$.

Tant que $i \leq j$, remplacer X_j par `ex*`$(X_j, b, 0)$ *et décrémenter j.*

Si $j < 0$ ou `C` $= 0$, *rendre X.*

Sinon positionner `C` *dans X_j, poser* `C` $= 0$, *et $i_B = j$. Rendre X.*

Procédure itimes0. *Paramètres X, Y et Z* [Il s'agit de calculer $XY + Z$ dans le vecteur Z. Les parties utiles de X et Y sont entre i_A et t_A pour Y, i_B et t_B pour X. Le chiffre le plus à droite de Z est à la position t_C. On positionne i_C sur le premier chiffre non nul du résultat. On suppose $t_C \geq t_A - i_A + t_B - i_B$. Si l'inégalité est stricte, il ne peut y avoir d'overflow. Dans le cas contraire, il sera dans `C`. La partie de Z qui sert pour la somme a la même taille que X].

Poser $k_0 = t_C - t_A$, $j = t_A$, $n = k_0 - (t_B - i_B + 1)$ et `C` $= 0$.

Tant que $j > i_A$

Si $Y_j = 0$, poser $k = j + n$.

Sinon, poser $i = t_B$, $k = k_0 + j$. Tant que $i \geq i_B$, remplacer Z_k par la quantité `ex*`(X_i, Y_j, Z_k), *décrémenter i et k.*

Décrémenter j.

Si $j \geq i_A$ mettre `C` *dans Z_k.*

Si `C` $= 0$ *ou $k < 0$, poser $i_C = k + 1$, dans le cas contraire, poser $Z_k =$* `C`, *puis* `C` $= 0$ *et $i_C = k$.*

Rendre Z.

2.4 Division

Division d'un nombre par un chiffre.

Les choses se compliquent un peu, car l'algorithme de division est non trivial. Commençons par un algorithme qui divise un nombre par un chiffre.

Algorithme 4. (Division par un chiffre) *On utilise un nombre interne x, un vecteur de chiffres X, un chiffre y et un indice i.* [Toutes les procédures rendent le quotient et mettent le reste dans la variable `C`].

Procédure nquoc. *Paramètres x et y.*

Poser `C` $= 0$.

Si $y = 1$, rendre x.

Si x est un chiffre, rendre `ex/`(x, y).

Si x n'a que deux chiffres, et que le premier est $< y$, poser `C` $= x_0$, *et rendre* `ex/`(x_1, y).

Sinon appeller `iquoc0` *sur x et y, puis* `underflow` *sur le résultat en entier.*

Procédure iquoc0. *Données X et y* [Suppose X non nul].

Poser $i_B = 0$, $t_B = t(X) - 1$.

Copier X.

Rendre `iquoc`(X, y).

Procédure iquoc. *Données X et y* [Diviser X entre i_B et t_B, suppose $i_B < t_B$, i_B indice du premier chiffre du quotient, mis à jour par la procédure].

Poser `C` $= 0$, *et $i = i_B$.*

Si $y = 1$ rendre X.

Dans le cas contraire, remplacer X_i par `ex/`(X_i, y) *et incrémenter i. Si le quotient est nul, poser $i_B = i$.*

Tant que $i \leq t_B$, remplacer X_i par `ex/`(X_i, y) *et incrémenter i.*

Rendre X.

Division de deux grands nombres.

Ceci est maintenant le gros algorithme de division. Le but est de diviser un nombre u par un nombre v. On exclut les cas triviaux : d'une part on suppose que v a au moins deux chiffres, et d'autre part $u > v > 0$.

On suppose que la partie utile de u est entre i_A et t_A, celle de v entre i_B et t_B. Le vecteur v ne sera pas modifié. Par contre on suppose que $i_C < i_A$, et la partie du vecteur u entre i_C et t_A sera modifiée : le quotient sera cadré à gauche, en commençant à l'indice i_C, le reste sera cadré à droite, en se terminant à l'indice t_A. Entre le quotient et le reste, il n'y a que des zéros. On modifie i_B et t_B de telle sorte que ce seront les indices utiles du reste. De plus, on modifiera t_A de sorte que ce sera l'indice du dernier chiffre du quotient. La condition $i_C < i_A$ est suffisante pour mettre le quotient et le reste dans le vecteur u. Elle est parfois nécessaire.

Analysons le problème. On va supposer $u = u_n + u_{n-1}E + \ldots + u_0E^n$, et noter cela par $u = (u_0u_1 \ldots u_n)$. On suppose que v a $m+1$ chiffres. A l'itération j on divise $m+2$ chiffres de u par v, les chiffres entre $j-1$ et $j+m$ (par convention $u_{-1} = 0$).

On veut obtenir le chiffre d'indice j du quotient. Il s'agit de résoudre

$$(u_{j-1}u_j \ldots u_{j+m}) = q_j(v_0v_1 \ldots v_m) + (r_0r_1 \ldots r_m) \tag{1}$$

donc

$$u_{j-1}E^2 + u_jE + u_{j+1} + \alpha = q_j(v_0E + v_1 + \beta) + r_0E + r_1 + \gamma \tag{2}$$

avec

$$0 \leq \alpha < 1 \qquad 0 \leq \beta < 1 \qquad 0 \leq \gamma < 1.$$

Ecrivons $u_{j-1}E + u_j = qv_0 + r$ par division. On a donc

$$rE + u_{j+1} + \alpha = v_0E(q_j - q) + q_j(v_1 + \beta) + r_0E + r_1 + \gamma. \tag{3}$$

On a $q_j \leq q$: comme $r < v_0$, le membre de gauche est $\leq (v_0-1)E+u_{j+1}+\alpha < (v_0-1)E+E-1+1 = v_0E$, le membre de gauche est donc $< v_0E$, et si $q_j > q$ le membre de droite est $> v_0E$.

On va maintenant supposer $E/e \leq v_0 < E$ (hypothèse de normalisation). Ajoutons eEv_0 aux deux membres de (3). Le membre de gauche devient $\geq E^2$. A droite on a $v_0E(q_j-q+e+1)+q_j(v_1+\beta)+E(r_0-v_0)+r_1+\gamma$. Supposons $q_j < q-1$. En particulier $q_j \leq E-1$, $q_j(v_1+\beta) < E(E-1)$, donc $q_j(v_1+\beta)+r_1+\gamma < E^2$. Comme $r_0 \leq v_0$, le membre de droite est $< E^2+v_0E(q_j-q+e+1)$. On en déduit $q_j > q-e-1$.

On a donc $q-e \leq q_j \leq q$. Ceci est une excellente estimation de l'erreur, surtout si $e=2$.

Supposons maintenant $q = q_j$. Alors (3) s'écrit

$$rE + u_{j+1} - v_1q_j + \alpha = q_j\beta + r_0E + r_1 + \gamma. \tag{4}$$

Le membre de droite est ≥ 0, d'où déduit $rE+u_{j+1} \geq v_1q_j$. A contrario, si $rE+u_{j+1} < v_1q$ c'est que q est trop grand. Posons alors $r' = r+v_0$, $q' = q-1$. On a la relation:

$$r'E + u_{j+1} + \alpha = v_0E(q_j-q') + q_j(v_1+\beta) + r_0E + r_1 + \gamma. \tag{5}$$

C'est la même relation que (3).

Itérons cette procédure de diminution de q jusqu'à avoir

$$rE + u_{j+1} \geq v_1q. \tag{6}$$

Supposons que $q_j = q-k$ avec $k > 0$. Alors (en reportant (6) dans (3)) $q_j\beta - kv_1 + r_0E + r_1 + \gamma \geq v_0Ek$. Or si $r_0 \leq E-2$, comme $q-k \leq E$ et $r_1+\gamma < E$, le membre de gauche est $< E^2$. Si $r_0 = E-1$, on a $r_0 = v_0$, donc $r_1 \leq v_1$ et $r_1 - kv_1 \leq 0$. En d'autres termes, $k < e$ (hypothèse de normalisation).

En particulier, dans le cas qui nous concerne, comme $e=2$, on a $q = q_j$ ou $q = q_j - 1$.

Décrivons maintenant le code qui permet de calculer q et r satisfaisant (6) (voir l'algorithme qui suit). On suppose que j est l'indice dans le vecteur u, les deux quantités u_j et u_{j+1} sont positionnées dans U_j et W, tandis que u_{j-1} est dans r. Les deux quantités v_0 et v_1 dont dans V_0 et V_1. On va d'abord calculer $u_{j-1}E + u_j = qv_0 + r$. Pour pouvoir utiliser **ex/** il faut que $u_{j-1} < v_0$. Dans le cas contraire, on aurait $q = E$ et $r = u_j$. Comme on sait que $q_j < E$, on sait a priori que q est faux. On va poser dans ce cas $q = 0$, $r = u_j$ et positionner une variable *spec* à vrai pour indiquer ce problème. Le calcul initial de q et r se fait à l'étape 2.2.

Dans le cas où la variable *spec* est faux, on va vérifier si (6) est vraie. On calcule pour cela $qv_1 = \alpha E + \beta$. Si $r > \alpha$ alors (6) est vraie, si $r < \alpha$ alors (6) est fausse, et si $r = \alpha$, (6) est vraie si $u_{j+1} \geq \beta$. Ce calcul est fait en 2.3.1. S'il faut décrémenter q, on le fait, et on calcule $r' = r+v_0 = AE+B$. Supposons $A = 0$, c'est donc que $r' = B$. Supposons $A = 1$. Le membre de gauche de (5) est alors $\geq E^2$. Le membre de droite est $\leq v_0E(q_j-q'+1)+q_j(v_1+\beta)+r_1+\gamma$. Si $q_j < q'$, il serait $\leq q_j(v_1+\beta)+r_1+\gamma \leq (E-1)(v_1+\beta)+r_1+\gamma < E^2$, absurde. On a donc $q_j = q'$ dans ce cas. Ce test est fait en 2.3.2. La quantité A est gardée dans la variable *OVF*.

Supposons $q = 0$. Comme on sait $0 \leq q_j \leq q$, c'est que q est exact. De plus il n'y a pas de soustraction à faire. Dans le cas contraire, faisons la soustraction.

$$(u_{j-1} \dots u_{j+m}) - q(v_0v_1 \dots v_m) = (r'_0r'_1 \dots r'_m).$$

Le calcul se fait de la façon suivante : soit C_k la retenue courante, initialisée par $C_m = 0$. On écrit

$$r'_k + qv_k + C_k = EC_{k-1} + u_{j+k}, \tag{7}$$

ou encore

$$C_k + qv_k + (E - 1 - u_{j+k}) = EC_{k-1} + (E - 1 - r'_k). \tag{8}$$

La relation (8) est utilisée dans l'algorithme à l'étape 2.4.2. On calcule le membre de gauche ce qui donne le membre de droite sous la forme $EC_{k-1} + x$. Si $y = E - 1 - x$, alors $E - 1 - y = x$, donc y est r'_k. La relation (7) est utilisée pour comprendre ce que l'on fait. Posons $y = C_0$, multiplions (7) par E^{m-k} et sommons. Il vient

$$(r'_1 \dots r'_m) + q(v_1 \dots v_m) = (u_{j+1} \dots u_{j+m}) + (y0 \dots 0).$$

En comparant avec (3) on obtient

$$Er + r'_1 + \gamma' = (q_j - q)(Ev_0 + v_1 + \beta) + (r_0 + y)E + r_1 + \gamma. \tag{9}$$

Supposons $q = q_j$. Alors $r'_i = r_i$ pour $i > 0$, et $\gamma = \gamma'$. Il reste $Er = (r_0 + y)E$, donc $r = r_0 + y$, $r_0 = r - y$, et $r \geq y$.

Dans le cas contraire, on sait (pour $e = 2$), que $q_j = q - 1$, donc

$$Er + r'_1 + \gamma' + Ev_0 + v_1 + \beta = (r_0 + y)E + r_1 + \gamma. \tag{10}$$

On en déduit $r + v_0 \leq r_0 + y$. Or, on sait $r_0 \leq v_0$, donc $r \geq y$. Mais, si $r_0 = v_0$ (i.e. $r = y$), on a $r_1 \leq v_1$, et (10) se réduit à $r'_1 + \gamma' + v_1 + \beta = r_1 + \gamma$, absurde, car $r_1 < v_1$ ou $r_1 = v_1$ et dans ce cas $\gamma < \beta$.

On en déduit le critère suivant : $q = q_j$ si et seulement si $r \geq y$. Calculons alors $1 + (E - 1 - y) + r = Ea + b$ (étape 2.4.5). Si $r \geq y$, on a $a = 1$, $b = r - y$, d'où $r_0 = b$. Donc si $a = 1$, c'est que $q = q_j$, $r'_i = r_i$ pour $i > 0$ et $r_0 = b$. L'étape de division est donc terminée. Remarquons que si $OVF = 1$, la vraie quantité r est en fait $E + B$, et l'on calcule $1 + (E - 1 - y) + B = r - y = r_0$. Comme on l'a déja noté, dans ce cas q est correct.

Supposons maintenant $a = 0$, et posons $r'_0 = E - y + r$. Alors (10) devient

$$Er'_0 + r'_1 + \gamma' + Ev_0 + v_1 + \beta = E^2 + Er_0 + r_1 + \gamma.$$

En d'autres termes, les r_i se calculent en additionnant v et r'. La retenue finale est à ignorer (à cause du terme E^2).

La fin de l'algorithme est simple : on va incrémenter j. Le contenu de u_j va devenir u_{j-1}, on le met donc dans r. Pour éviter tout problème dans l'algorithme de Bezout, on va mettre 0 dans u_j (ne pas oublier de remettre r dans le vecteur après la dernière division). Il reste finalement qu'à s'assurer que le quotient est bien cadré à gauche, et à calculer les indices.

Algorithme 5. (Division interne, procédure iquomod8.) *Données u et v. On utilise comme variables et paramètres* V_0, V_1, U_j, W, OVF, r, q *et* l, *des chiffres,* N, j, i_q, M, L *et* k, *des indices,* first, spec *et* ok *des booléens. Les deux données u et v sont des vecteurs de chiffres.* [La sémantique de l'algorithme est donnée au début de la section.]

1. *Poser* $N = t_B - i_B$, $j = i_A$, $r = 0$, $i_q = i_C$, $m = t_B$, $V_0 = v_{i_B}$, $V_1 = v_{i_B+1}$, $M = t_A - N$, first = *vrai*, spec = *faux*.

2. *Répéter* $M - i_A + 1$ *fois*

2.1. *Poser* $U_j = u_j$ *et* $W = u_{j+1}$. *Poser* ok = *faux*, OVF = 0.

2.2.a. *Si* $r = V_0$, *poser* $q = 0$, $r = U_j$, spec = *vrai*.

2.2.b. *Dans le cas contraire, poser* $\texttt{C} = r$, $q = \texttt{ex/}(U_j, V_0)$ *et* $r = \texttt{C}$.

2.3. *Tant que* ok *est faux* [Première correction du quotient.]

2.3.1. *Si* spec *est faux, alors : poser* $\texttt{C} = 0$, *soit* $l = \texttt{ex*}(q, V_1, 0)$. *Positionner* ok *à vrai si* $\texttt{C} < r$, *à faux si* $\texttt{C} > r$, *et si* $\texttt{C} = r$, *à vrai si* $l \leq W$.

2.3.2. *Si* spec *est vrai ou si* ok *est faux, décrémenter* q, *positionner* spec *à faux,* C *à 0,* r *à* $\texttt{ex+}(r, V_0)$ *et* OVF *à* C. *Si* C *est non nul, mettre* ok *à vrai.*

2.4. *Si* q *est non nul* [soustraction]

2.4.1. *Poser* $L = m$, $k = N + j$, $\texttt{C} = 0$.

2.4.2. *Tant que* $L > i_B$, *remplacer* u_k *par* $\texttt{ex-}(\texttt{ex*}(v_L, q, \texttt{ex-}(u_k)))$, *décrémenter* L *et* k.

2.4.3. *Soit* $l = \texttt{ex-}(\texttt{C})$.

2.4.5. *Poser* $\texttt{C} = 1$. *Remplacer* u_j *par* $\texttt{ex+}(r, l)$.

2.4.6. *Si* $\texttt{C} \neq 0$ *et* OVF *est 0* [seconde correction du quotient]

Décrémenter q.

Poser $L = m$, $k = N + j$, $\texttt{C} = 0$.

Tant que $L \geq i_B$, *remplacer* u_k *par* $\texttt{ex+}(u_k, v_L)$, *décrémenter* k *et* L.

2.5. *Poser* $r = u_j$, *puis* $u_j = 0$.

2.6. *Si* first $=$ *vrai et* $q = 0$, *ne rien faire. Sinon, poser* first $=$ *faux, positionner* q *dans* u *à la position* i_q, *et incrémenter* i_q.

2.7. *Incrémenter* j.

3. *Poser* $u_{j-1} = r$.

4. *Poser* $t_B = t_A$, $t_A = i_q - 1$, *et* $i_B = \texttt{afind0-beg}(u, M, t_B)$.

5. *Rendre* u.

Quotient par un nombre de deux chiffres.

Dans le cas où on divise par un nombre de deux chiffres, on peut optimiser. On fait exactement les mêmes hypothèses que pour `iquomod8`. Les optimisations sont les suivantes : d'une part les boucles internes d'addition et de soustraction ont été supprimées (la soustraction est une boucle de longueur 1, l'addition de longueur 2). D'autre part, on ne va pas chercher à réaliser la condition (6). En effet, l'étape de division consiste à résoudre

$$CE^2 + U_1E + U_2 = q(V_0E + V_1) + r_0E + r_1.$$

Supposons $C < V_0$. Ecrivons $CE + U_1 = qV_0 + r$, puis $qV_1 = b_1E + a_1$, puis $(E - 1 - a_1) + U_2 + 1 = \alpha E + r_1$ et $(E - 1 - b_1) + r + \alpha = \beta E + r_0$. Alors

$$(1 - \beta)E^2 + CE^2 + U_1E + U_2 = q(V_0E + V_1) + r_0E + r_1. \tag{11}$$

Si $\beta = 1$, alors q est le quotient, et $Er_0 + r_1$ est le reste. On sait en effet que q est exact ou trop grand. Si $Er_0 + r_1$ n'est pas le reste, c'est qu'il est plus grand, absurde. Dans le cas $C = V_0$, on a la même relation avec $q = E$, $\beta = 0$, si $r_1 = U_2$ et $r_0 = (E - 1 - V_1) + U_1 + 1$. Ce calcul est fait en 2.2.a ou 2.2.b.

Supposons $\beta = 0$, donc q trop grand. Ecrivons $r_1 + V_1 = \alpha E + r_1'$ puis $r_0 + V_0 + \alpha = \beta E + r_0'$, et $q' = q - 1$. On a donc

$$q'(V_0E + V_1) + r_0'E + r_1' = \beta E^2 + q(V_0E + V_1) + r_0E + r_1,$$

ce qui donne la relation (11) avec des primes là où il faut.

Répétons donc l'opération tant que β est nul. On suppose qu'avant l'opération q est trop grand. Donc après, q est exact ou trop grand. Ceci nous donne par conséquent l'algorithme suivant (on a essayé de mettre les mêmes numéros que pour l'algorithme précédent).

Algorithme 6. (Division par deux chiffres, two-div) *Données u et v. On utilise comme variables locales et paramètres u et v, des vecteurs de chiffres, U_1, U_2, V_0, V_1, q, r, a_1, b_1, r_0 et r_1, des chiffres, i et j des indices et un booléen* first.

1. *Poser $i = i_A$, $j = i_C$,* first = vrai. *Soient V_0 et V_1 les deux chiffres de v, $\mathtt{C} = 0$ et $U_1 = u_i$.*

2. *Répéter $t_A - i_A$ fois*

2.1. *Incrémenter i, poser $U_2 = u_i$.*

2.2.a. *Si $\mathtt{C} = V_0$, poser $q = 0$, $\mathtt{C} = 1$, $r_1 = U_2$ puis $r_0 = \mathtt{ex{-}{-}}(U_1, V_1)$.*

2.2.b. *Sinon poser $q = \mathtt{ex/}(U_1, V_0)$, $r = \mathtt{C}$, $\mathtt{C} = 0$, $a_1 = \mathtt{ex*}(q, V_1, 0)$, $b_1 = \mathtt{C}$, $\mathtt{C} = 1$, $r_1 = \mathtt{ex{-}{-}}(U_2, a_1)$ et $r_0 = \mathtt{ex{-}{-}}(r, b_1)$.*

2.4.6. *Tant que $\mathtt{C} = 0$, décrémenter q, poser $r_1 = \mathtt{ex+}(r_1, V_1)$ et $r_0 = \mathtt{ex+}(r_0, V_0)$.*

2.6. *Si* first = vrai *et $q = 0$, ne rien faire. Sinon, poser first =* faux, *positionner q dans u à la position j, et incrémenter j.*

2.8. *Poser $U_1 = r_1$ et $\mathtt{C} = r_0$.*

3. *Mettre r_0 dans u en position $i - 1$, et r_1 en position i.*

4. *Poser $t_B = t_A$, $t_A = j - 1$, et $i_B = i - 1$ si r_0 est non nul, i si r_0 est nul mais pas r_1, et $i + 1$ si les deux sont nuls.*

5. *Rendre u.*

Division générale.

Maintenant que nous avons fait le plus gros du travail, nous pouvons expliquer l'algorithme de division. Cet algorithme rend le quotient et positionne le reste dans la variable globale **mod** (en fait **#:ex:mod**).

La seule hypothèse que nous allons faire en divisant x par y est que y est non nul. On va donc commencer par comparer x et y, car dans le cas $x = y$, le quotient est 1 et le reste est 0, et dans le cas $x < y$, le quotient est 0 et le reste est x. Si y est un chiffre, on utilise la procédure **nquoc** décrite précédemment.

Supposons donc que x et y sont des vecteurs de chiffres, entre 0 et t_1 pour x, entre 0 et t_2 pour y. Supposons de plus que le chiffre en position s de y est non nul, et qu'il n'y a que des 0 après. En d'autres termes y est YE^m. Ecrivons $x = XE^m + Z$. Ecrivons alors $X = QY + R$ par division euclidienne. Alors le quotient de x par y est Q et le reste est $RE^m + Z$.

Supposons d'abord $s = 0$, un seul chiffre dans Y, et $t_1 = t_2$, un seul chiffre à considérer dans X. Le quotient et le reste Q et R sont donc triviaux. On est cependant astucieux en ce qui concerne la gestion de la mémoire pour le calcul du reste r.

Supposons maintenant $s = 0$ et $t_1 \neq t_2$. La division va se faire par **iquoc**. On calcule i_B et t_B, les indices à considérer pour **iquoc**. Si **iquoc** rend un reste non nul, le vrai reste est dans le

vecteur X entre les indices $t_B + 1$ et t_1. On ne connait pas le nombre exact de chiffres du reste, il faut le calculer. Par contre, si R est non nul, le nombre de chiffres du reste est connu. On va extraire le quotient du vecteur X, y positionner R, puis extraire le reste (procédure `arrange-q0`).

Dans le cas général on a $s > 0$. On va appeler l'algorithme général de division sur X et Y. Notons que cet algorithme suppose que Y est normalisé. Soit N le facteur de normalisation. On va alors écrire $NX = Q'(NY) + R'$. En d'autres termes, $Q = Q'$ et $R = R'/N$. Comme multiplier X par N peut augmenter sa taille et que l'algorithme de division demande un chiffre de libre, on copie X (en fait x) dans un vecteur plus long de deux cases (donc de taille $t_1 + 3$). Comme la multiplication va écraser Y, on copie Y (en fait y) si $N \neq 1$.

On positionne alors les quantités suivantes : i_A et t_A les indices de X dans la copie de x, i_B et t_B les indices de Y dans y (ou sa copie), et t_C le dernier indice de x dans sa copie.

On va alors appeler `iquomod4`. Cette procédure est coupée en deux pour des raisons purement techniques. Elle va positionner Q et R dans le vecteur X, le quotient est entre les indices i_A et t_B, et le reste entre les indices i_B et t_B. Elle positionne t_B à l'ancien t_A. En ce qui nous concerne, X est x, et juste derrière se trouve Z. En d'autres termes, le reste r est le contenu de x entre les indices i_B et t_C.

La procédure commence par multiplier X et Y par N. Bien entendu, dans le cas $N = 1$, on ne fait rien. La multiplication se fait via `itimesc0`, et on fait attention aux indices i_A, etc, qui peuvent être modifiés. La vraie division se fait par `iquomod8` ou `two-div` suivant que Y a plus de deux chiffres ou non. Finalement, si N n'est pas 1, on divise le reste R' par N via `iquoc`. Il n'y a pas lieu de s'occuper des variables d'indices (tout est fait pour), il faut juste vérifier que le reste est non nul.

Algorithme 7. (Division des entiers) *On utilise comme variables locales et paramètres x et y, des entiers, X et Y des vecteurs de chiffres, N et q des chiffres, et t_1, t_2, s, i et j des indices.*

Procédure nquomod. *Paramètres x et y.*

1. *Si $x = y$ poser* `mod` $= 0$, *rendre 1.*

2. *Si $x < y$ poser* `mod` $= x$, *rendre 0.*

3. *Si y est un chiffre, calculer $q =$* `nquoc`(x, y). *Poser* `mod` $=$ `C` *et rendre q.*

4. [Ceci est la division en général] *Dans les autres cas, la procédure est comme suit.*

4.1. *Commencer par calculer les tailles t_1 et t_2 de x et y. Leur soustraire 1.*

4.2. *Soit $s =$* `afind0-end`(y).

4.2.1. *Si $s = 0$, et $t_1 = t_2$, rendre* `iquomod2`(x, y, t_1).

4.2.2. *Si $s = 0$ et $t_1 \neq t_2$, copier x, poser $t_B = t_1 - t_2$, $i_B = 0$, et $t_C = t_1$. Appeler* `iquoc` *sur x et y_0, puis* `arrange-q0` *sur le résultat.*

4.2.3. *Si $s > 0$, soit N le facteur de normalisation de y. Si N n'est pas 1, copier y.*

4.2.3. *Poser $i_B = 0$, $t_B = s$, $t_A = 2 + t_1 - (t_2 - s)$, $i_C = 0$, $i_A = 2$, $t_C = t_1 + 2$.*

4.2.3. *Copier x dans un vecteur de taille $t_1 + 3$, cadré à droite.*

4.2.3. *Appeler* `iquomod4` *sur x, y et N.*

4.2.3. *Mettre* `nunderflow`(x, i_B, t_C) *dans* `mod`.

4.2.3. *Rendre* `nunderflow`(x, i_A, t_A).

Procédure iquomod2. *Paramètres X, Y et s.* [on suppose $X = X_0 E^s$ et $Y = Y_0 E^s$].

Poser `C` $= 0$, $q =$ `ex/`(X_0, Y_0).

Si `C` $\neq 0$, *copier* X, *mettre* `C` *dans* X_0, *et mettre* X *dans* `mod`.

Sinon, mettre `nunderflow`$(X, 1, s)$ *dans* `mod`.

Rendre q.

Procédure arrange-q0. *Données* X.

Si `C` $\neq 0$, *allouer un vecteur de taille* $t_C - t_B + 1$. *En position 0, y mettre* C, *y copier derrière* X *à partir de l'indice* $t_B + 1$. *Mettre le résultat dans* `mod`.

Si $C = 0$, *mettre* `nunderflow`$(X, t_B + 1, t_C)$ *dans* `mod`.

Rendre `nunderflow`(X, i_B, t_B).

Procédure iquomod4. *Paramètres* X, Y *et* N.

Si $N = 1$ *appeler* `iquomod8t` *sur* X, Y *et* N.

Sinon, appeler `itimesc0` *sur* Y, N *et 0.*

Poser $i = i_B$ *et* $j = t_B$.

Poser $t_B = t_A, i_B = i_A$. *Appeler* `itimesc0` *sur* X, N *et 0. Poser* $t_A = t_B, i_A = i_B$.

Poser $t_B = j, i_B = i$.

Appeler `iquomod8t` *sur* X, Y *et* N.

Procédure iquomod8t *Paramètres* X, Y *et* N.

Appeler `two-div` *ou* `iquomod8` *suivant que* Y *a ou non 2 chiffres.*

Poser $i_A = i_C$.

Si $i_B \leq t_B$ *et* N *différent de 1, appeler* `iquoc` *sur* X *et* N.

Rendre X.

2.5 Impression

Le premier algorithme que nous allons donner imprime un entier en base b. Cette base peut n'avoir aucun rapport avec la constante $E = e^p$. On suppose $2 \leq b \leq 36$. Ceci est dû au fait qu'on suppose qu'un chiffre en base b est l'un des 10 chiffres entre 0 et 9, ou l'une des 26 lettres de l'alphabet. On suppose savoir imprimer un chiffre $< b$ (fonction **princn** de Lisp). On suppose également que l'imprimeur Lisp sait imprimer des petits entiers. La procédure que nous allons décrire ici n'imprime donc que des grands entiers. Elle ne teste pas si le nombre tient sur la ligne courante : les caractères sont imprimés les uns après les autres, si la ligne courante est pleine, un changement de ligne est généré automatiquement. La stratégie utilisée par SISYPHE pour imprimer des grands entiers est simplement d'imprimer ces nombres dans le tampon d'impression (nettoyé au préalable) puis d'extraire le résultat du tampon, et d'imprimer ceci comme une chaîne de caractères (ou suite de chaînes). Il est également possible d'utiliser la fonction **explode**, elle est moins efficace car elle rend la liste des codes ASCII.

Tout le problème est donc de calculer les chiffres de notre nombre de façon optimale, en évitant au maximum les divisions. Pour des raisons d'efficacité, le code de la division est une copie du code de `iquoc` (de plus codé en LLM3).

On suppose que dans une table `table` se trouve à la position i une liste (x, j) où $x = i^j$, et j est la plus grande puissance telle que $x \leq E$. On va donc essayer d'obtenir j chiffres d'un coup, par division. Comme le reste du nombre par la base fournit le dernier chiffre à imprimer, il faut ranger ces nombres dans une table. On vérifie si $e = 2$ et $p = 16$, qu'une table de taille $3/2$ fois la taille du nombre est assez grande. Enfin, dans le cas où $x = E$, on met 0 à la place de x dans la table et on ne fait pas de division du tout. On utilise une table `buf16` de taille q telle que $2^{q+1} > E$ (pour toute base, tout nombre $< E$ a au plus q chiffres dans cette base). Ce vecteur est de taille 16 si $E = 2^{16}$.

Algorithme 8. (Impression des entiers) *On utilise comme paramètres ou variables, x, z et q des chiffres, i, j, J et k, des indices n un nombre interne et X et Y, des vecteurs de chiffres.*

Procédure nprin0, procédure principale. *Paramètre n.*

Si n est un chiffre, appeler `aprin-no-just`.

Sinon, soit b la base, (z, j) le contenu de `table` *à l'indice b.*

Si $z \neq 0$, appeler `iprin0` *sur n, z et j.*

Sinon

Soit k la taille de n et $i = 1$.

Appler `aprin-no-just` *sur n_0.*

Tant que $i < k$, appeler `aprin-just` *sur n_i et j et incrémenter i.*

Procédure aprin-no-just. *Paramètres x* [$x < E$, imprimer x].

Si $x = 0$, imprimer le chiffre 0.

Sinon, soit $i = 0$, tant que $x \neq 0$ poser `C` $= 0$, $x =$ `ex/`(x, b), *mettre* `C` *dans* `buf16` *en position i, et incrémenter i.*

Appeler `output-print-vect` *sur i.*

Procédure aprin-just. *Paramètres x et j* [$x < E$, imprimer x avec j chiffres].

Soit $i = 0$, tant que $i < j$ poser `C` $= 0$, $x =$ `ex/`(x, b), *mettre* `C` *dans* `buf16` *en position i, et incrémenter i.*

Appeler `output-print-vect` *sur i.*

Procédure output-print-vect. *Paramètre i* [Imprimer les i premiers chiffres de `buf16`].

Tant que $i > 0$ décrémenter i, et imprimer en base b le chiffre en position i de `buf16`.

Procédure iprin0. *Paramètres X, z et j.*

Copier X.

Allouer un vecteur Y de taille $3s/2$ où s est la taille de X. Poser $i = 0$, $J = 0$ et $k = 0$.

Faire [boucle principale de calcul des chiffres]

Poser $i = J$, `C` $= 0$, $q =$ `ex/`(X_i, z), *remplacer X_i par q, et incrémenter i.*

Si $q = 0$, poser $J = i$.

Tant que $i < s$ remplacer X_i par `ex/`(X_i, z) *et incrémenter i.*

Mettre `C` *dans Y en position k, incrémenter k.*

Fin de la boucle si $J = s$.

Décrémenter k.

Appeler `aprin-no-just` *sur Y_k et décrémenter k.*

Tant que $k \geq 0$, appeler `aprin-just` *sur Y_k et j, et décrémenter k.*

Autre algorithme d'impression. Cet algorithme imprime les p premiers chiffres d'une fraction en base b. Si le nombre est périodique, la période est entre accolades. Par exemple, $19/6$ sera imprimé comme `3.1{6}`. Les chiffres à gauche du point s'obtiennent en imprimant la partie entière du quotient n/d, donc q si $n = dq + r$.

Il s'agit alors d'imprimer la partie fractionnaire. Supposons que n et d soient premiers entre eux. C'est cette condition qui va nous permettre de trouver la période. Soit x le premier chiffre imprimé, et n'/d' ce qui reste à imprimer après le premier chiffre. On a donc la relation

$$\frac{n}{d} = \frac{x}{b} + \frac{n'}{bd'}.$$

Cette relation s'écrit aussi $nb = dx + dn'/d'$. On veut $0 \leq x < b$ et $0 \leq dn'/d' < d$. En d'autres termes, x est le quotient de la division de nb par d; si $bn = xd + r$, on a $d'r = dn'$. Comme on veut que d' soit premier avec n', c'est que d' divise d, donc $d = \lambda d'$. On a donc $r = \lambda r'$. Or $bn = \lambda(d'x + r')$, donc λ divise b, donc μ le pgcd de b et d. Réciproquement, si $\lambda = \mu$, et si on calcule $nb/\mu = x(d/\mu) + n'$, alors n' et $d' = d/\mu$ sont premiers entre eux. Comme la base b est au plus 36, on préfère calculer le pgcd de b et d, plutôt que de calculer le pgcd de d et r.

Notons qu'à l'itération suivante, on calculera $\mathrm{pgcd}(b, d/\mu)$. Si p est un nombre premier divisant ce pgcd, et premier à μ, il divisera d et b, donc μ, absurde. On en déduit que ce pgcd sera trivial à partir du moment où $\mu = 1$. En fait, on divise d par toutes les puissances des facteurs premiers de b jusqu'à obtenir à un certain moment $\mathrm{pgcd}(b, d) = 1$.

Ce phénomène se produit nécessairement (sauf si n s'annule, auquel cas le nombre est complètement imprimé). Supposons que N est la valeur de n au moment où le pgcd μ devient 1. Ce que l'on fait ensuite est de calculer $bn = dx + r$, et de remplacer n par r. Au bout de k itérations n est remplacé par $b^k N$ modulo d. Comme b et N sont premiers à d, cette quantité n'est jamais nulle. De plus, b est un élément du groupe inversible modulo d, lequel groupe est de cardinal $\phi(d)$. Donc $b^{\phi(d)} = 1$ modulo d. En particulier, la suite des n (donc la suite des x) est périodique, de période $\phi(d)$. La plus petite période peut bien entendu être un diviseur strict de $\phi(d)$, par exemple $7/9$ est `0.{7}` en base 10, `0.{61}` en base 8, `0.{530}` en base 7, et `0.{342102}` en base 5, et $\phi(9) = 6$.

Si la période commençait avant que μ ne devienne 1, on aurait les relations suivantes. Soit x le dernier chiffre de la période. On a $bn' = dx + r'$, et $r' = N$. On fait l'hypothèse que $n''b/\mu = x(d''/\mu) + r''$, et que $d''/\mu = d$ et que $r'' = N$. On en déduit $n'\mu = n''$, donc μ divise n''. Or μ divise d'', μ est non trivial, et n'' et d'' sont premiers entre eux, absurde. On sait donc localiser le début de la période. Ceci fournit l'algorithme suivant.

Algorithme 9. (Ecriture décimale) *Données n, d et p. Variables locales q, r, et N. Les arguments et variables locales sont des entiers positifs ou nuls* [On imprime le quotient n/d avec au plus p chiffres après la virgule].

1. *Ecrire $n = qd + r$ par division euclidienne, imprimer q, et poser $n = r$.*

2. *Imprimer un point.*

3. *Poser* $q = \mathrm{pgcd}(b, d)$, $B = b/q$, *et* $d = d/q$.

4. *Faire*

4.1. *Si* $q = 1$, *fin de la boucle.*

4.2. *Ecrire* $nB = dq + r$, *imprimer* q, *poser* $n = r$.

4.3 *Poser* $q = \mathrm{pgcd}(b, d)$, $B = b/q$, *et* $d = d/q$.

4.4 *Décrémenter* p. *Fin de la boucle si* $p = 0$ *ou* $n = 0$.

5. *Si* $n = 0$, *fin de l'algorithme.*

6. *Si* $p = 0$, *imprimer trois petits points, fin de l'algorithme.*

7. *Poser* $N = n$, *imprimer une accolade ouvrante.*

8. *Tant que* $p > 0$ *faire*

8.1. *Ecrire* $nB = dq + r$, *imprimer* q, *poser* $n = r$.

8.2. *Décrémenter* p.

8.3. *Si* $n = N$, *poser* $p = 0$, *sinon si* $p = 0$, *imprimer trois petits points.*

9. *Imprimer une accolade fermante.*

2.6 Pgcd

On propose l'algorithme suivant de calcul de pgcd.

Algorithme 10. (Pgcd sur les entiers positifs) *On utilise comme variables locales et paramètres* a *et* b, *des nombres internes,* x, y, N, *des chiffres,* u, v *des vecteurs de chiffres, et* i, j *des indices.*

On utilise une variable globale Z *qui est un tableau de chiffres.*

Procédure npgcd, fonction principale. *Paramètres* a *et* b.

Comparer a *et* b.

Si $a = b$, *rendre* a.

Si $a < b$ *appeler* `ipgcd3` *sur* b *et* a.

Sinon appeler `ipgcd3` *sur* a *et* b.

Procédure ipgcd3. *Données* a *et* b [On suppose $a > b$].

Si a *est un chiffre rendre* `ipgcd0`(a, b).

Si b *est un chiffre, poser* $i_B = 0$, $t_B = t(a) - 1$, *copier* a, *et rendre* `ipgcd1`(a, b).

Sinon, soit $s = t(a) + 2$. *Allouer un vecteur* Z *de taille* s. *Copier* a *et* b *dans des vecteurs de taille* s, *cadrés à droite. Poser* $i_A = 2$, $t_A = s - 1$, $t_B = s - 1$, $i_B = s - t(b)$. *Appeler* `ipgcd2` *sur* a *et* b.

Procédure ipgcd0. *Données* x *et* y.

Si $y = 1$, rendre 1.

Poser `C` $= 0$. *Calculer* `ex/`(x, y). *Si* `C` $= 0$, *rendre* y, *sinon* `ipgcd0`$(y,$ `C`$)$.

Procédure ipgcd1. *Données u et y.*

Appeler `iquoc` *sur u et y.*

Si `C` $= 0$, *rendre* y, *sinon* `ipgcd0`$(y,$ `C`$)$.

Procédure ipgcd2. *Données u et v* [On suppose $u > v$].

Appeler `igcd-via-bezout` *sur u et v.*

Echanger u et v [On suppose alors $u > v$.]

Si $t_B < i_B$, rendre `nunderflow`(u, i_A, t_A).

Si $t_B = i_B$, poser $y = v_{i_B}$, $i_B = i_A$, $t_B = t_A$, et rendre `ipgcd1`(u, y).

Sinon rendre `ipgcd2`(u, v).

Procédure igcd-via-div. *Données u et v.*

Poser $i = i_B$ et $j = t_B$, $i_C = 0$.

Soit N le facteur de normalisation de v.

Si $N \neq 1$, copier v dans Z.

Appeler `iquomod4` *sur u, v (ou Z si on a copié) et N.*

Poser $i_C = t_A$.

Poser $i_A = i$ et $t_A = j$.

Algorithme de Lehmer.

La première implémentation de cet algorithme consistait à utiliser `igcd-via-div` au lieu de `igcd-via-bezout`. Ce que nous allons décrire est la nouvelle version (algorithme de Lehmer).

L'idée de base est que très souvent le quotient calculé est petit. On estime ce quotient en prenant les premiers bits, et on fait plusieurs divisions successives sur ces premiers bits. Au bout de quelques opérations, u est remplacé par $Au+Bv$, et v est remplacé par $Cu+Dv$. Ceci revient à faire 4 multiplications d'un grand entier par un chiffre et deux soustractions. L'algorithme de division calcule essentiellement quatre multiplications et une soustraction (une multiplication interne, deux multiplications pour la normalisation et une division, qui équivaut à une multiplication). Bien entendu, lors de la division, la soustraction et la multiplication se font en une passe, mais on peut faire de même dans ce cas. Si on arrive donc à remplacer plusieurs divisions par une opération de ce type on y gagne. Si on arrive à remplacer une division par une opération de ce type on y gagne souvent (le quotient est connu, on peut éviter la phase de normalisation). Bien entendu, s'il faut quand même faire une division on y perd, mais pas trop.

Notons que les divers quotients apparaissant lors du calcul du pgcd sont les quotients du développement en fraction continue du quotient. Dans [5, §3.6] cette technique est utilisée pour compacter les représentations des réels sous forme de fraction continue.

La première chose à faire est de calculer les premiers bits de u et v, en d'autres termes x et y tels que $u = e^k(x + \hat{x})$ et $v = e^k(y + \hat{y})$, où x et y sont des entiers et $0 \leq \hat{x} < 1$, $0 \leq \hat{y} < 1$. On veut $0 < x < E$, en fait on veut x le plus grand possible. Cette méthode n'a aucun intérêt si l'on

prend pour x les bits qui se trouvent dans le chiffre principal de u. On veut donc $E/e \leq x < E$. Supposons que u soit dans le vecteur A entre les indices i_A et t_A. Comme on est dans la procédure **ipgcd2**, on sait que $i_A < t_A$. Supposons que les chiffres en position i_A et $i_A + 1$ de u soient α et β. On suppose que les chiffres aux mêmes indices de v sont α' et β'. Dans le cas où $i_A < i_B$, on prend $\alpha' = 0$ (en fait, on met 0 dans v à la position i_A, cela servira dans la suite). Dans le cas où $i_A + 1 < i_B$ on a de plus $\beta' = 0$. Comme nous allons le voir dans la suite, ce cas est sans intérêt, car il faut faire une division. Quitte à décrémenter i_B, on supposera donc $i_B = i_A$. Notons aussi que $t_B = t_A$, dans la mesure où on a pris la précaution de mettre u et v dans des vecteurs de même taille, cadrés à droite. Une telle condition restera toujours vraie. Cette façon de faire a comme avantage de ne pas être obligé de mettre à jour les indices t_A et t_B.

Soit alors i le nombre de bits de α. Comme on veut p bits, il faut en rajouter $p - i$. Soit $w = e^i$, et divisons $E\alpha + \beta$ par w. Comme $E\alpha + \beta$ a $p + i$ bits, le quotient aura p bits, et sera x. De même le quotient de $E\alpha' + \beta'$ par w sera y. Notons que si $i = p$, on a tout simplement $x = \alpha$ et $y = \beta$. Ces calculs sont faits par la procédure **iget-first** dans l'algorithme qui suivra.

Appelons x_0 et y_0 les quantités x et y ainsi calculées. Posons $n = 0$. On considère des quantités A, B, C et D, (initialisées avec 1, 0, 0 et 1) ainsi que $A' = (-1)^n A$, $B' = -(-1)^n B$, $C' = -(-1)^n C$ et $D' = (-1)^n D$. Posons $x' = x + B$, $y' = y + D$, $x'' = x + A$ et $y'' = y + C$. On va supposer que ces quantités sont entre 0 et $E - 1$ (sauf éventuellement x'' et y' si $n = 0$). Une telle relation est évidente pour $n = 0$. Notons que $x'' = E$ si $x = E - 1$, et $y' = E$ si $y = E - 1$. Mais, comme $0 \leq y \leq x$, on aura aussi $x = y$. Comme nous allons le voir dans un instant, ce cas est sans intérêt, on l'excluera donc. En fait, si $x = y$, les deux nombres u et v commencent par p bits identiques. Il y a donc intérêt à remplacer u par $u - v$, et ceci est plus efficace que de faire une division. Le seul cas où une de ces quantités est E est donc $n = 0$, et $x = E - 1$.

Posons $\alpha = uA + vB$ et $\beta = uC + vD$. On a donc $u = \alpha$ et $v = \beta$ pour $n = 0$. On va supposer que, si on faisait effectivement les n divisions sur u et v, les quantités u et v seraient transformées en α et β. Pour deviner les quotients de ces divisions, on suppose que la partie entière de α/e^k est entre x' et x'', celle de β/e^k est entre y' et y''. Comme nous allons le montrer dans la suite, les quantités A', B', C' et D' sont positives ou nulles, donc, si n est pair, on a $x' \leq x \leq x''$ et $y'' \leq y \leq y'$, les inégalités sont dans l'autre sens si n est impair.

Ecrivons $x' = q'y' + r'$ et $x'' = q''y'' + r''$ par division euclidienne. Si l'une des deux quantités y' ou y'' est nulle, le quotient sera l'infini (avec le signe adéquat). On a donc $q' \leq \alpha/\beta < q'' + 1$, donc si $q' = q''$, c'est la partie entière de α/β. Dans le cas où il n'y a pas égalité, on ne connait pas le quotient, et on s'arrête. En particulier, si q' ou q'' est l'infini, on s'arrête. On ne manipule donc pas vraiment l'infini.

Supposons donc $q = q' = q''$. On pose alors $A_1 = C$, $C_1 = A - qC$, $B_1 = D$, $D_1 = B - qD$, $x_1 = y$ et $y_1 = x - qy$. Les quantités x'_1, etc. s'en déduisent. En particulier, $A'_1 = C'$, $C'_1 = A' + qC'$, $B'_1 = D'$, $D'_1 = B' + qD'$, ce qui montre que ces quatre quantités sont positives ou nulles. Nous allons les borner dans la suite, de telle sorte que l'addition-multiplication se fait via **ex***. Notons que y_1 est aussi le reste de la division de x par y, ce qui fait qu'il n'y a pas de soustraction à faire.

Notons que $x'_1 = y'$, $x''_1 = y''$, $y'_1 = r'$, et $y''_1 = r''$. Ces relations montrent que ces quantités sont entre 0 et E. En fait, elles ne peuvent être E. Le seul cas où cela pourrait l'être est si $x'_1 = y' = E$. Ce cas a été exclus, car en effet, cela impose $n = 0$, $x = y = E - 1$. Or $x''/y'' = E/(E-1) > 1$ et $x'/y' = (E-1)/E < 1$. En d'autres termes $q' \neq q''$, ce qui justifie le fait d'arrêter l'algorithme de suite. On peut faire le même raisonnement si $x = y$. Supposons que pour $n = 0$ on ait $y = 1$. On calcule donc $x'/y' = x/2$ et $x''/y'' = (x+1)/1$, et les deux quotients ne sont pas le mêmes, on s'arrête de suite. Finalement, $\alpha_1 = \beta$ et $\beta_1 = \alpha - q\beta$. Des relations $x' \leq \alpha/e^k \leq x''$ et $y'' \leq \beta/e^k \leq y'$ (valides pour n pair) on tire $y'_1 = x' - qv' \leq \beta_1/e^k \leq x'' - qv'' = y''_1$. Le même raisonnement s'applique si n est impair, en renversant le sens des inégalités.

On supprime alors les indices 1, on incrémente n, et on continue. On vient donc de calculer A, B, C, et D tels que $\alpha = uA + vB$ et $\beta = uC + vD$, avec $\beta < \alpha$. Notons que $x = x_0A + y_0B$ et $y = x_0C + y_0D$. Cette relation s'écrit aussi $x_0 = D'x + B'y$ et $y_0 = C'x + A'y$ (voir l'algorithme de Bezout général où les quantités A', B', C' et D' s'appellent x, y, u et v). Comme ces deux relations sont entre des nombres positifs ou nuls, on en tire des inégalités importantes. D'abord, y est non nul, car sinon, y' et y'' auraient des signes opposés, donc l'un au moins serait nul (ils sont positifs ou nuls). Donc C ou D est nul. Or D' est strictement croissant, il est non nul, et on a $C = 0$. On en déduit $y_0 = 0$. Or ce cas a été exclus a priori (la procédure se termine sans avoir fait aucune division, il faut utiliser `igcd-via-div`).

Comme après la première itération on a $y < x$, on en déduit $x \geq 2$, et C' et D' sont majorés par $E/2$. Il en est de même de A' et B' (quantités à l'itération précédente).

Il reste maintenant à expliquer comment calculer une quantité de la forme $uA + vB$. En fait, il faut calculer $\pm(uA' - vB')$. Montrons donc comment calculer $uA' - vB'$. On sait que A' et B' sont des chiffres, u et v des vecteurs de chiffres. Supposons d'abord $A' = 1$. Il s'agit de calculer $u - vB'$. On utilise la même procédure utilisée par `iquomod8` pour la soustraction interne. Dans le cas $A' \neq 1$, on calcule $-vB'$ de la même manière, en faisant comme si u était nul. Il s'agit alors de calculer $uA + v'$, ce qui est facile. En fait, on réalise les deux opérations en une seule boucle, en manipulant deux retenues. Finalement, dans le cas $B' = 1$, on calcule $-v$. Ce calcul est trivial : en allant de droite à gauche, on laisse les 0 inchangés. Si v_i est le premier chiffre non nul, on le remplace par $E - v'$. Les autres chiffres sont remplacés par $E - 1 - v_i$.

Finalement, on procède comme suit. Soit n le nombre de divisions faites. Si $n = 0$, et $x = y$, on remplace u par $u - v$. Si $n = 0$ mais $u \neq v$, on divise via `igcd-via-div`. Si $n = 1$, le quotient est D, et on remplace u par $u - Dv$. Sinon on remplace u par $uC + vD$ et v par $uA + vB$. Comme ces opérations sont censées être faites en parallèle, on met le résultat de la première dans Z, le résultat de la seconde dans v, et on copie Z dans u (notons que Z est un vecteur auxilliaire, utilisé pour copier v s'il faut faire une division et normaliser v). Ceci donne l'algorithme suivant.

Algorithme 11. (pgcd via Lehmer) *Les variables et paramètres utilisés ici sont a et b, des nombres internes, x, y, N, A, B, C, D, α, β, α', β', c, c', n, d, q, n', d', q', s, w et t des chiffres, u, v et z, des vecteurs de chiffres et i, j et k, des indices. On utilise de plus la variable globale Z un vecteur de chiffres. Ces variables sont celles utilisées ici et dans l'algorithme global de calcul du pgcd.*

Procédure iget-first. *Paramètres u et v.*

1. *Si $i_A + 1 < i_B$, rendre $x = 1$ et $y = 0$. Sinon, si $i_A < i_B$ mettre 0 dans v en position i_A.*

2. *Calculer α et β les éléments en position i_A et $i_A + 1$ de u, α' et β' les éléments en position i_A et $i_A + 1$ de v.*

3. *Soit k le nombre de bits de α et $w = e^k$.*

4.a. *Si $k = p$, poser $x = \alpha$ et $y = \beta$.*

4.b. *Sinon, poser* `C` $= \alpha$, $x =$ `ex/`(β, w), `C` $= \alpha'$, $y =$ `ex/`(β, w).

5. *Rendre x et y.*

Procédure iget-ABCD. *Paramètres x et y.*

1. *Poser $s = 1$, $A = 1$, $B = 0$, $C = 0$, $D = 1$, $N = 0$.*

2. *Si $x = y$, poser $N = 1$. Si $x = y$ ou si $y = 0$ ou $y = 1$, rendre A, B, C, D, s et N.*

3. *Faire*

3.a.1. *Si* $s = 1$

Poser $\mathtt{C} = 1$, $d = \mathtt{ex{-}{-}}(y, C)$.
Si $d = 0$, *fin de la boucle.*
Poser $\mathtt{C} = 0$, $n = \mathtt{ex{+}}(x, A)$, $q = \mathtt{ex/}(n, d)$.
Poser $\mathtt{C} = 0$, $d' = \mathtt{ex{+}}(y, D)$.
Si $d' = 0$, *fin de la boucle.*
Poser $\mathtt{C} = 1$, $n' = \mathtt{ex{-}{-}}(y, B)$, $\mathtt{C} = 0$ *et* $q' = \mathtt{ex/}(n', d')$.

3.a.2. *Si* $s = -1$

Poser $\mathtt{C} = 0$, $d = \mathtt{ex{+}}(y, C)$.
Si $d = 0$, *fin de la boucle.*
Poser $\mathtt{C} = 1$, $n = \mathtt{ex{-}{-}}(x, A)$, $\mathtt{C} = 0$, $q = \mathtt{ex/}(n, d)$.
Poser $\mathtt{C} = 1$, $d' = \mathtt{ex{-}{-}}(y, D)$.
Si $d' = 0$, *fin de la boucle.*
Poser $\mathtt{C} = 0$, $n' = \mathtt{ex{+}}(x, B)$ *et* $q' = \mathtt{ex/}(n', d')$.

3.b. *Si* $q \neq q'$, *fin de la boucle.*

3.c. *Poser* $q' = A$, $A = C$, $\mathtt{C} = 0$, $C = \mathtt{ex*}(q, C, q')$.

3.d. *Poser* $q' = B$, $B = D$, $D = \mathtt{ex*}(q, D, q')$.

3.e. *Calculer* $\mathtt{ex/}(x, y)$, *poser* $x = y$, $y = \mathtt{C}$.

3.f. *Remplacer* s *par* $-s$ *et incrémenter* N.

4. *Rendre* A, B, C, D, s *et* N.

Procédure igcd-via-bezout *Paramètres* u *et* v.

1. *Calculer* x *et* y *via* `iget-first`.

2. *Appeler* $\mathtt{iget\text{-}ABCD}(x, y)$. *Ceci rend* A, B, C, D, s *et* N.

3.a. *Si* $N = 0$, *appeler* `igcd-via-div` *sur* u *et* v.

3.b. *Si* $N = 1$, *appeler* $\mathtt{isubstract}(u, v, D)$.

3.c. *Sinon*

3.c.1. *Si* $s = 1$, *Poser* $i_B = \mathtt{isub}(v, D, u, C, Z)$ *sinon* $i_B = \mathtt{isub}(u, C, v, D, Z)$.

3.c.2. *Si* $s = 1$, *Poser* $i_A = \mathtt{isub}(u, A, v, B, v)$, *sinon* $i_A = \mathtt{isub}(v, B, u, A, v)$.

3.c.3. *Copier* Z *dans* u.

4. *Rendre* A, B, C, D, s, N.

Procédure isub *Paramètres* u, x, v, y *et* z [Calcule $ux - vy$, le résultat est dans z, les chiffres utiles sont entre i_A et t_A].

1. *Poser* $i = t_A$.

2.a. *Si* $x = 1$, *poser* $\mathtt{C} = 0$, *tant que* $i \geq i_A$ *remplacer* z_i *par* $\mathtt{ex\text{-}}(\mathtt{ex*}(v_i, y, \mathtt{ex\text{-}}(u_i)))$ *et décrémenter* i.

2.b. *Si* $y = 1$, *poser* $s = 0$, *tant que* $i \geq i_A$,

Si $s = 1$, *poser* $\mathtt{C} = 0$, $\alpha = \mathtt{ex\text{-}}(v_i)$.
Si $v_i = 0$ *poser* $\alpha = 0$.
Sinon, poser $s = 1$, $c = \mathtt{C}$, $\mathtt{C} = 0$, $\alpha = \mathtt{ex{-}{-}}(1, v_i)$ *et* $\mathtt{C} = c$.

Remplacer z_i *par* `ex*`(u_i, x, α) *et décrémenter* i.

2.c. *Sinon poser* $c = c' = 0$. *Tant que* $i \geq i_A$ *faire*

Poser `C` $= c'$, $\alpha =$ `ex-(ex*`$(v_i, y, -1)$`)`, $c' =$ `C`.

Poser `C` $= c$, *remplacer* z_i *par* `ex*`(u_i, x, α), *poser* $c =$ `C`.

Décrémenter i.

3. *Rendre* `afind0-beg`(z, i_A, t_A).

Procédure isubstract *Paramètres* u, v *et* x [Calcule $u - vx$, le résultat est dans u, les chiffres utiles sont entre i_A et t_A].

Poser `C` $= 0$ *(1 si* $x = 1$*)*, $i = t_A$.

Tant que $i \geq i_A$, *remplacer* u_i *par* `ex-(ex*`$(v_i, x,$`ex-`$(u_i))$) *(ou si* $x = 1$ *par* `ex--`(u_i, v_i)*) et décrémenter* i.

Poser $i_A = i_B$ *et* $i_B =$ `afind0-beg`$(u, i + 1, t_A)$.

Temps de calcul.

Nous donnons dans la table qui suit le temps de calcul du pgcd de F_n et F_{n+1} où F_n est le n-ième nombre de Fibonacci. Ce temps est donné en secondes CPU sur Sparc Station 2. Quelques remarques : tous les quotients successifs sont 1, ce qui signifie que l'algorithme de Lehmer s'applique de façon optimale. On peut en déduire, que, en moyenne les deux algorithmes, celui de Sisyphe et celui de bigNum qui repose sur l'arithmétique de [1] sont équivalents. L'avant-dernier calcul dans la version de Lisp ayant pris 5 heures de CPU, on peut estimer que le dernier calcul prend près de 7 heures. Il n'a pas été effectué.

n	sisyphe	BN	Lisp	BN/sis	Lisp/sis	AKCL	Maple	Pari
1000	0.08	0.55	41	6.8	512	2.2	0.06	0.08
2000	0.23	1.10	267	4.8	1160	9.8	0.23	0.28
3000	0.47	1.71	588	3.6	1250	20.5	0.50	0.59
4000	0.79	2.49	1845	3.2	2340	37.3	0.90	1.01
5000	1.15	3.09	2927	2.7	2550	59.0	1.37	1.56
6000	1.62	3.87	4073	2.4	2520	84.6	1.98	2.24
7000	2.17	5.40	8654	2.5	3990	115.3	2.67	3.05
8000	2.83	5.58	13364	2.0	4720	151.4	3.50	3.94
9000	3.51	6.44	18000	1.8	5130	193.0	4.44	4.93
9999	4.24	7.46		1.8		235.1	5.44	5.92

2.7 Bezout

Préliminaires

Etant donné deux entiers A et B, le but de cet algorithme est de trouver deux entiers u et v tels que $uA + vB = p$ où p est le pgcd de A et B. Nous supposerons ici $A > b > 0$. La fonction utilisateur ne fait bien entendu aucune hypothèse sur A et B.

Le calcul de la relation de Bezout implique le calcul du pgcd de A et B, et est une généralisation de l'algorithme d'Euclide. Posons $a_0 = A$, $b_0 = B$, et écrivons par récurrence $a_n = b_n q_n + r_n$, puis $a_{n+1} = b_n$, $b_{n+1} = r_n$, comme lors du calcul du pgcd.

Introduisons la matrice $Q_n = \begin{pmatrix} q_n & 1 \\ 1 & 0 \end{pmatrix}$, et quatre quantités x_n, y_n, u_n et v_n. Posons

$$\overline{M}_n = \begin{pmatrix} v_n & y_n \\ u_n & x_n \end{pmatrix} \qquad C_n = \begin{pmatrix} a_n \\ b_n \end{pmatrix} \tag{1}$$

avec $\overline{M}_0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ comme condition initiale pour $n = 0$ et $C_0 == \begin{pmatrix} A \\ B \end{pmatrix}$.

La relation de récurrence donnant les a_n et les b_n s'écrit simplement

$$Q_n C_{n+1} = C_n. \tag{2}$$

Les quantités x_n, y_n, u_n et v_n sont définies par les relations $x_{n+1} = u_n$, $y_{n+1} = v_n$, $u_{n+1} = x_n + q_n u_n$, $v_{n+1} = y_n + q_n v_n$. Ce sont des quantités positives ou nulles. Les formules s'écrivent aussi sous la forme matricielle

$$\overline{M}_{n+1} = \overline{M}_n Q_n. \tag{3}$$

En combinant les relations (2) et (3) on voit que la matrice $\overline{M}_n C_n$ ne dépend pas de n. On en déduit

$$a_n v_n + b_n y_n = A \tag{4.a}$$

$$a_n u_n + b_n x_n = B \tag{4.b}$$

d'où l'on déduit que le pgcd de a_n et b_n divise le pgcd de A et B.

Comme la matrice Q_n est de déterminant -1, on a $x_n v_n - y_n u_n = (-1)^n$ donc 'inverse de $\overline{M}_n$ est $M_n = (-1)^n \begin{pmatrix} x_n & -y_n \\ -u_n & v_n \end{pmatrix}$ Les relations (4) s'écrivent donc aussi

$$(-1)^n x_n A - (-1)^n y_n B = a_n \tag{5.a}$$

$$-(-1)^n u_n A + (-1)^n v_n B = b_n \tag{5.b}$$

ou plus simplement $M_n C_0 = C_n$.

Une conséquence triviale de ces relations est que le pgcd de A et B divise le pgcd de a_n et b_n, donc lui est égal. Considérons M tel que b_M divise a_M. C'est le pgcd de A et B. Alors (5.b) s'écrit

$$-(-1)^M u_M A + (-1)^M v_M B = \text{pgcd}(A, B). \tag{6}$$

Cette relation est appelée relation de Bezout, et l'algorithme que nous allons donner va calculer les différentes quantités. Nous allons faire cependant deux optimisations importantes.

Optimisations.

Supposons que $M' = (-1)^m \begin{pmatrix} \alpha & -\beta \\ -\gamma & \delta \end{pmatrix}$ soit une matrice de déterminant $(-1)^m$ et supposons $C_{n+m} = M' C_n$ i.e.

$$a_{n+m} = (-1)^m a_n \alpha - (-1)^m b_n \beta \tag{7.a}$$

$$b_{n+m} = -(-1)^m a_n \gamma - (-1)^m b_n \delta \tag{7.b}$$

$$\alpha\beta - \gamma\delta = (-1)^m. \tag{7.c}$$

On a ce genre de relations si on utilise l'algorithme de Lehmer, ou si on applique l'algorithme de Bezout à a_n et b_n. De la relation $C_{n+m} = M'C_n$ on tire $M_{n+m} = M'M_n$. En d'autres termes :

$$x_{n+m} = \alpha x_n + \beta u_n \tag{8.a}$$

$$y_{n+m} = \alpha y_n + \beta v_n \tag{8.b}$$

$$u_{n+m} = \gamma x_n + \delta u_n \tag{8.c}$$

$$v_{n+m} = \gamma y_n + \delta v_n. \tag{8.d}$$

Estimons maintenant les quantités x, y, u et v. Comme $a_0 > b_0$, et $a_{n+1} > b_{n+1}$ (définition de la division euclidienne), on aura $q_n \geq 1$ pour tout n. On en déduit $v_{n+1} \geq v_n$. En particulier $v_n \geq 1$, d'où $y_n \geq 1$ pour $n > 0$, donc $v_{n+1} > v_n$ pour $n > 0$. En évaluant (4.a) en $n = M$, on voit que la suite des y_i et v_i est majorée par A/p donc par A si p est le pgcd cherché. De la même manière la suite des x_n est croissante (à partir de $n = 1$, car $x_0 = 1$ et $x_1 = 0$).

La seconde amélioration de l'algorithme consiste à ne calculer qu'une seule des deux quantités u_M, v_M, de préférence la plus petite u_M et à calculer l'autre via la relation (6). En fait, ceci n'est pas tout à fait évident. Nous avons constaté sur un exemple que la division finale prenait environ la moitié du temps de calcul de toutes les diverses quantités y_n et v_n. Calculer v_M par la boucle prend plus de temps (on travaille sur des nombres plus grands), par contre, la division finale se fait sur des nombres plus petits.

Détails de l'algorithme

Considérons d'abord les relations (8). Il s'agit de calculer $x_1 = \alpha x + \beta u$ et $u_1 = \gamma x + \delta u$, sachant que x et u sont des vecteurs de chiffres. On suppose que le premier indice de x est i_D et celui de u est i_E. Pour simplifier, on suppose que les deux vecteurs ont même taille. Cette taille sera dans la variable t_A, ce sera aussi la taille de a_n et b_n (cette façon de faire minimise le nombre de variables globales utilisées). On suppose également qu'un vecteur z peut être utilisé pour ranger des résultats intermédiaires. Evidemment, on veut mettre x_1 dans x et u_1 dans u en parallèle.

Notons que $x = (x_1 - \beta u)/\alpha$, donc $u_1 = (\gamma x_1 + (\alpha\delta - \beta\gamma)u)/\alpha$. Comme $\alpha\delta - \beta\gamma = \pm 1$, ceci se réduit à $u_1 = (\gamma x_1 \pm u)/\alpha$. On peut donc se passer de variables intermédiaire. Nous estimons cependant que la division est moins efficace que la multiplication, et qu'il est relativement plus efficace de calculer x_1 dans z, puis u_1 dans u et de copier z dans x. La différence de temps ne doit pas être en fait tellement grande.

La procédure **`ibezout-add`** a pour but de calculer $y = \alpha x + \beta u$. Comme $\alpha\delta - \beta\gamma = \pm 1$, les deux quantités α et β ne peuvent être simultanément nulles. On traite à part le cas où l'une des deux est nulle (en fait, l'autre ne peut être que 1 dans ce cas, mais cette procédure peut être appelée par ailleurs). On traite également à part le cas où l'une des deux quantités est 1, car le principe est le même que pour **`itimesc0`**.

Considérons le cas général. Pour i variant de i_A à i_D, on calcule $x_i\alpha + u_i\beta + c$ sous la forme $Ec' + z_i$ où c est la retenue entrante et c' la retenue sortante. Notons que $0 \leq c \leq 2E - 3$, et toutes les valeurs sont permises. Or $2E - 3 < E$ uniquement si $E = 2$, cas qui ne nous intéresse pas. Il faut donc gérer deux retenues. Pour cela, on écrit $x_i\alpha + c + \mathtt{C} = \mathtt{C}'E + p$, puis $\beta u_i + p = c'E + z_i$. A la fin de la boucle, il suffit d'additionner les deux retenues.

Relation de Bezout entre des chiffres.

On calcule U, V, n et p tels que $-Ux + Vy = (-1)^n p$, où p est le pgcd de x et y. De plus M est incrémenté de n. Notons x_1 et y_1 les valeurs initiales de x et y. Supposons $u' = 0$ et $x' = 1$, de sorte que y_1 divise $x'x_1 - (-1)^n x$ et $u'x_1 + (-1)^n y$, au moins pour $n = 0$. Ces deux relations de divisibilité forment l'invariant de l'algorithme. Tant que y ne divise pas x, on écrit $x = qy + r$, on remplace x par y, y par r, et on incrémente n. On calcule les nouvelles valeurs de x' et u'. A la fin de la boucle, le pgcd est y. On sait que $-u'x_1 + Vy_1 = (-1)^n y$ pour un certain $V \geq 0$. Le calcul de V dépend de la parité de n.

Cas d'un nombre a et d'un chiffre b.

Ecrivons $a = bq + r$. Dans le cas où le reste est nul, le pgcd est b, $M = 0$ et $U = 0$, c'est un cas trivial. Dans le cas contraire, écrivons $-Ub + Vr = (-1)^n p$ grâce à la méthode proposée précédemment. On a donc $aV - b(U + qV) = (-1)^n p$. Il n'y a donc pas de calculs supplémentaires à faire.

Cas de deux nombres u et v.

Sauf cas particuliers, les deux quantités U et X auront à la fin à peu près la même taille que u et v, ce seront des vecteurs de chiffres. On commence par copier u et v dans des vecteurs de taille t, la taille de u plus 2 (comme pour le calcul du pgcd). On alloue deux vecteurs de chiffres de même taille U et X, et un vecteur auxilliaire Z. Celui-ci servira de tampon à la fois pour la division et pour l'addition.

Dans une première phase (procédure `ibezout3`), on appelle `igcd-via-bezout`. Ceci a comme effet de faire N divisions, et modifie u et v. On suppose que B divise $AX - (-1)^n u$ et $AU + (-1)^n v$. Dans le cas $N = 0$, on a remplacé u par le quotient et le reste de la division par v, le reste est entre i_B et t_B, le quotient entre 0 et i_C. De plus i_A et t_A sont les indices dans v. Dans le cas où le reste est nul, le pgcd est v et l'on a $-AU + BV = (-1)^n u$, où V n'est pas calculé. On a fini. Dans le cas où le reste est non nul, il s'agit de remplacer X par U et U par $Uq + X$. On utilise `itimes0` pour ce faire, en faisant attention aux indices. Il ne faut pas oublier d'incrémenter n.

Dans le cas où $N = 1$, on sait que le quotient a un seul chiffre. On pourrait utiliser `itimesc0` pour calculer $Uq + X$, mais on préfère `ibezout-add` car cela simplifie la gestion des indices.

Finalement, dans le cas $N \geq 2$, on fait plus d'une division. On utilise les relations (8.a) et (8.b).

Dans une seconde phase (procédure `ibezout2`), après la division, on regarde les nouvelles valeurs de u et v. Dans le cas où v a au moins deux chiffres, on boucle. Dans le cas où v n'a aucun chiffre (est nul) on a fini. Il reste à regarder le cas où v a un seul chiffre. Si u a plus d'un chiffre, on fait comme dans le cas général, sauf qu'on utilise `iquoc` au lieu de `igcd-via-div` pour faire la division. On se ramène donc au cas où u et v ont un seul chiffre. On calcule leur relation de Bezout, et on utilise également les relations (8).

Algorithme 12. (Relation de Bezout) *On utilise beaucoup de variables.*

Procédure nbezout, fonction principale. *Données a, et b* [On suppose $a > b > 0$].

1. *Poser* $M = 0$.

2.a. *Si a est un chiffre, rendre* `ibezout0`(a, b).

2.b. *Si b est un chiffre*

Calculer `iquoc0`(a, b).

Si `C` $= 0$, *poser* $U = 0$ *et rendre* b.

Sinon, poser $p =$ `ibezout0`$(b,$ `C`$)$, *incrémenter* M, *Poser* $U = V$ *et rendre* p.

2.c. *Cas général.*

Soit $s = t(a) + 2$, *poser* $i_A = 2$, $i_B = s - t(B)$. *Allouer des vecteurs* U, X *et* Z *de taille* s.

Copier a *et* b *dans des vecteurs de taille* s, *cadrés à droite.*

Mettre 1 en position $s - 1$ *dans* X, *poser* $i_D = s - 1$ *et* $i_E = s$.

Rendre `ibezout2`(a, b).

Procédure ibezout0. *Données* x *et* y [calcule et rend le pgcd p, de plus U et V tels que $-Ux + Vy = (-1)^m p$, met à jour M].

1. *Poser* $n = 0$, `C` $= 0$, $x_1 = x$, $y_1 = y$, $u' = 0$, $x' = 1$.

2. *Faire* [boucle de division]

Poser $q =$ `ex/`(x, y), $r =$ `C`.

Si $r = 0$, *fin de la boucle.*

Poser $x = y$, $y = r$, `C` $= 0$, $r =$ `ex*`(q, u', x'), $x' = u'$ *et* $u' = r$.

Incrémenter n.

3. *Poser* $M = M + n$.

4.a. *Si* n *est impair, poser* `C` $= 1$, $q =$ `ex*`$(u',$ $x_1,$`ex-`$(y))$, *décrémenter* `C`, *et poser* $V =$ `ex/`(q, y_1)

4.b. *Sinon, poser* $V =$ `ex/`$($`ex*`$(u', x_1, y), y_1)$

5. *Poser* $U = u'$, *et rendre* y.

Procédure ibezout2. *Paramètres* u *et* v.

1. *Appeler* `ibezout3` *sur* u *et* v, *échanger* u *et* v.

2.a. *Si* $i_B < t_B$, *rendre* `ibezout2`(u, v).

2.b. *Si* $i_B > t_B$ *remplacer* U *par* `nunderflow`(X, i_D, t_B), *décrémenter* M, *rendre la quantité* `nunderflow`(u, i_A, t_A).

2.c.1. *Sinon, poser* $r_1 = v_{i_B}$ [seul chiffre de v] *et* $r_r = u_{i_A}$ [premier chiffre de u].

2.c.2. *Si* $i_A < t_A$ [u a plus d'un chiffre]

Poser $i_B = i_A$, *calculer* `iquoc`(u, r_1).

Si `C` $= 0$ *remplacer* U *par* `nunderflow`(U, i_E, t_B) *et rendre* r_1.

Sinon incrémenter M, *poser* $r_2 = r_1$ *et* $r_1 =$ `C`.

Poser $A = U$, $t_C = t_A$, $k = i_E$, $i_A = i_B$, $t_A = t_B$, $i_B = \min(i_D, i_E)$, $t_C = t_B$.

Calculer `itimes0`(U, u, X).

Poser $U = X$, $X = A$, $i_D = k$ *et* $i_E = i_C$.

2.c.3 [On a donc deux fois un chiffre]

Poser $U' = U$, $X' = X$.

Poser $p =$ `ibezout0`(r_2, r_1).
Poser $k =$ `ibezout-add`(X', U, U', V, Z).
Poser $U =$ `nunderflow`(Z, k, t_B).
Rendre p.

Procédure ibezout3. *Paramètres* u *et* v [On divise via Lehmer].

1. *Appeler* `igcd-via-bezout` *sur* u *et* v. *Ceci rend* α, β, γ, δ, s *et* N.

2. *Si* $N = 0$,

2.1. *Si* $i_B > t_B$ *ne rien faire.*
2.2. *Poser* $A = U$, $k = i_E$, $i = i_B$, $j = i_A$,
2.3. *Poser* $i_B = \min(i_D, i_E)$ $i_A = 0$, $t_A = i_C$, $t_C = t_B$
2.4. *Poser* $U =$ `itimes0`(U, u, X), $i_E = i_C$, $i_B = i$, $i_A = j$, $t_A = t_B$
2.5. *Poser* $X = A$, $i_D = k$.
2.6. *Incrémenter* M.

3. *Si* $N = 1$,

3.1. *Si* $i_B > t_B$ *ne rien faire.*
3.2. *Poser* $A = U$, $k = i_E$
3.3. *Poser* $i_E =$ `ibezout-add`$(X, 1, U, \delta, X)$ *et* $U = x$.
3.4. *Poser* $X = A$, $i_D = k$.
3.5. *Incrémenter* M.

4. *Sinon,* $[n \geq 2]$

4.1. *Poser* $k =$ `ibezout-add`(X, α, U, β, Z).
4.2. *Poser* $i_E =$ `ibezout-add`$(X, \gamma, U, \delta, U)$.
4.3. *Copier* Z *dans* X, *poser* $i_D = k$.
4.4. *Incrémenter* M *de* n.
4.5. *Si* $i_B > t_B$, *décrémenter* M.

Procédure ibezout-add. *Paramètres* u, x, v, y *et* z [Calcule $ux + vy$, résultat dans z, entre les indices $\min(i_D, i_E)$ et t_A].

1. *Poser* `C` $= 0$, $j = \min(i_D, i_E)$.

2.a. *Cas* $x = 0, y = 1$. *Si* $z \neq v$, *copier* v *dans* z, *poser* $j = i_E$.

2.b. *Cas* $x = 1, y = 0$. *Si* $z \neq u$, *copier* u *dans* z, *poser* $j = i_D$.

2.c. *Si* $x = y = 1$, *pour* i *de* t_A *à* j, *remplacer* z_i *par* `ex+`(u_i, v_i).

2.d. *Si* $x = 1$ *pour* i *de* t_A *à* j, *remplacer* z_i *par* `ex*`(v_i, y, u_i).

2.e. *Si* $y = 1$ *pour* i *de* t_A *à* j, *remplacer* z_i *par* `ex*`(u_i, x, v_i).

2.f. *Poser* $c = 0$. *Pour* i *de* t_A *à* j, *poser* $q =$ `ex*`(u_i, x, c), $c =$ `C`, `C` $= 0$, *Remplacer* z_i *par* `ex*`(v_i, y, q).

2.f. *Si* $c \neq 0$, *poser* $c =$`ex+`$(c, 0)$. *Si* `C` *est nul, poser* `C` $= c$, *sinon, remplacer* z_i *par* c, *et décrémenter* j.

3.a. *Si* `C` $\neq 0$, *décrémenter* j, *remplacer* z_j *par* `C`, *et rendre* j.

3.b *Sinon, rendre* j.

Chapitre 3

Arithmétique rationnelle

3.1 Structures de données

Dans ce chapitre nous définissons les entiers et les rationnels et les opérations sur ces objets.

Un entier est soit un chiffre soit un vecteur. Dans le premier cas, il ne prend pas de place en mémoire, ce qui est important. On suppose que l'entier x est représenté par le chiffre c. Si $c < E/2$, ont dit que $x = c$, dans le cas contraire, on dit que $x = E - c$. On exclut en général la valeur $c = E/2$ (pour en parler il faut la nommer `#$8000`). Un grand entier est un vecteur de type `#:r:z` à deux champs. Le premier champ est le signe (vrai si le nombre est positif, faux sinon) et le second est un nombre interne, en général un vecteur, sauf dans le cas où $E/2 \leq |x| < E$.

Un rationnel n'est jamais un entier, c'est le quotient de deux entiers a et b, représenté sous forme d'un vecteur de type `#:r:q` à quatre champs. Le premier champ est le signe. Il est suivi par les représentations internes des valeurs absolues de a et b. Finalement le dernier champ est un indicateur, qui s'il est vrai indique que a et b sont premiers entre eux.

Par défaut, toutes les fractions sont systématiquement réduites. La variable-fonction **fractions-are-reduced** permet de consulter ou modifier un indicateur interne. Si celui-ci est faux, le calcul de pgcd n'est pas systématique. On peut dans ce cas réduire une fraction individuelle à l'aide de la fonction **reduce-fraction**. Cette fonction rend une nouvelle fraction, mais elle peut aussi modifier physiquement la fraction si la variable interne associée à la variable fonction **reduced-fractions-are-displaced** est vraie.

La fonction (interne) principale de création des fractions est `qx`. Elle prend en arguments les valeurs absolues des numérateurs et dénominateurs. Le signe de la fraction est dans la variable globale `sign1`. De plus, elle suppose positionné l'indicateur `is-simplified` : si la valeur est vraie, le numérateur et le dénominateur sont premiers entre eux. Dans le cas contraire, il faut calculer un pgcd. Ce pgcd n'est calculé que si la variable `reduce-fraction` est vraie. Si elle est fausse, on ne fait pas de calcul de pgcd. Il faut par contre s'assurer que le résultat n'est pas entier. Pour être plus efficace, la fonction suppose que la variable `is-not-integer` est positionnée : si elle est vraie, la fraction est garantie ne pas être un entier, si elle est fausse, on divise le numérateur par le dénominateur pour s'en assurer. Il y a plusieurs fonctions de création des entiers : par exemple `zx` crée un entier en faisant les mêmes hypothèses que `qx`, la fonction `mknumpos` crée un entier positif, etc.

3.2 Fonctions élémentaires sur les rationnels

Les deux fonctions génériques **numerator** et **denominator** rendent le numérateur et le dénominateur d'un nombre. Ces fonctions n'ont de sens que pour les nombres rationnels, en d'autres termes, provoquent une erreur d'intitulé (en anglais) "This operation has no meaning for its argument". Le numérateur d'un entier est l'entier lui-même, le dénominateur est 1 par convention. En ce qui concerne une fraction a/b, le numérateur est la quantité a (avec le signe de la fraction) est le dénominateur est b (positif). Ces deux fonctions réduisent systématiquement les fractions, contrairement à l'option choisie dans LeLisp, mais conforme à celle de Common Lisp (cf [4, p. 215].

La fonction générique **0-** existe pour tous les nombres définis dans SISYPHE. En ce qui concerne les flottants et les petits entiers, on laisse le soin à Lisp de le faire (c'est une des raisons pour lesquelles $E/2$ n'est pas considéré comme un petit entier, ce serait le seul à ne pas avoir d'opposé sous forme de petit entier, et son opposé serait le seul grand entier dont l'opposé serait un petit entier). En ce qui concerne les entiers et les rationnels, pour trouver l'opposé, il suffit juste de changer le signe. Finalement, pour prendre l'opposé d'un nombre complexe, il suffit de prendre l'opposé des parties réelle et complexe.

La fonction générique **1/** calcule l'inverse d'un nombre. Notons que l'inverse de 0 n'existe pas, les entiers 1 et -1 sont leur propres inverses. Pour tous les autres entiers x, leur inverse est $1/x$, le résultat n'est certainement pas entier, et la fraction est toujours réduite. L'inverse d'une fraction a/b est la fraction b/a. Si la fraction de départ est réduite, l'inverse l'est aussi, et le résultat ne peut être entier que si a est trivial. Dans le cas où la fraction n'est pas réduite, on ne peut rien dire a priori. Finalement l'inverse d'un nombre complexe $a + ib$ est le nombre $a/d - ib/d$ où le dénominateur d est $a^2 + b^2$.

3.3 Comparaison

La fonction générique **signum** rend le signe de son argument. Pour les nombres complexes, elle est définie par la formule $x/|x|$, le résultat est donc un nombre complexe. Dans les autres cas, le résultat est 0 si le nombre est nul, 1 s'il est positif, et -1 sinon (notons que le résultat est toujours entier, contrairement à la stratégie préconisée par Common Lisp).

La fonction **abs** rend la valeur absolue d'un nombre. Ceci est simple dans le cas des nombres réels. Par convention la valeur absolue d'un nombre complexe $a + ib$ est son module $\sqrt{a^2 + b^2}$. La racine carrée est calculée en utilisant l'arithmétique générique. Aucune optimisation n'est faite pour l'instant, donc la valeur absolue de $3 + 4i$ est le flottant 5.

Toutes les autres fonctions de comparaison utilisent l'arithmétique générique de Lisp, que ce soient les fonctions d'égalité ou d'inégalité, elles appellent la fonction <?>, qui rend 0, 1 ou -1 (par extension, elle peut rendre NIL, tous les prédicats rendant faux si le résultat de <?> n'est pas 0, 1 ou -1). Cette méthode est particulièrement mal adaptée dans le cas des complexes et des BigFloats (ces nombres ne sont pas décrits ici, mais une implémentation embryonnaire a été faite dans SISYPHE. Il s'agit de nombre réels de taille arbitraire, et de précision limitée. Si on ne connait deux nombres qu'à ϵ près, on ne peut pas toujours décider lequel est le plus grand). En ce qui concerne la comparaison d'un nombre complexe et d'un autre nombre, on rend 0 si les nombres sont égaux, () sinon. Avec cette façon de faire, l'égalité est bien définie. Par contre <> n'est plus la négation de =, en effet *(<> a b)* est systématiquement faux si l'un des deux nombres est complexe. L'arithmétique par défaut de Lisp rend un nombre complexe dont la partie réelle est le résultat des comparaisons des parties réelles, la partie imaginaire le résultat de la comparaison

des parties imaginaires. Ce mécanisme bizarre est tel que les fonctions de comparaisons de deux nombres rendent le résultat correct si la différence des deux nombres est réelle, elles rendent faux dans les autres cas. Nous avons préféré implémenter un mécanisme plus simple.

La fonction de comparaison est une fonction générique à deux arguments. Elle appelle une fonction dépendant du type du premier argument. Cette fonction regarde le type du second argument. Pour simplifier le code, on échange les arguments s'il le faut. On est amené à comparer un entier et un entier, un entier et un rationnel, ou deux rationnels. Si l'un des arguments est un nombre flottant, on convertit le second en flottant et l'on compare.

Comparaison de deux entiers a et b. Si les deux entiers sont des chiffres, la fonction interne de comparaison des chiffres fournit le résultat. Dans le cas contraire, quitte à échanger les deux nombres, on peut supposer que a est un grand entier. Dans le cas où b est un chiffre, on sait $|b| < |a|$, il suffit de rendre le signe de a. Si b est un grand entier, deux cas se présentent : si a et b n'ont pas le même signe, on rend le signe de a, si les deux nombres sont positifs, on compare les valeurs absolues, et s'ils sont négatifs, on compare les valeurs absolues dans l'autre ordre.

Comparer deux valeurs absolues (nombres internes) est simple, et fait par la fonction **ncmp** qui rend 0 si les deux nombres sont égaux, 1 si le premier est plus grand que le second et -1 sinon. En général ce sont des vecteurs de chiffres, si ces vecteurs n'ont pas la même taille, le plus long est le plus grand, et dans le cas contraire, on boucle sur les chiffres, en les comparant avec **ex?**.

La situation se complique pour la comparaison d'un rationnel x et d'un nombre y, entier ou rationnel. On teste d'abord les signes des deux nombres : s'ils sont distincts, la comparaison est triviale. On est amené au problème de comparer $x = \pm a/b$ (où le signe est dans une variable s) avec y (de même signe). On compare les valeurs absolues, puis, le cas échéant, inverse le résultat. Supposons d'abord que y soit entier. De la relation $a/b < a$ on sait que si $a \leq y$ alors $x < y$. On compare donc a et y. Si cela donne le bon résultat, on s'arrête, dans le cas contraire, on compare a et by. La même ruse est utilisée dans le cas où $y = c/d$ est une fraction : si $a < c$ et $b > d$ alors $x < y$ (s'il y a deux égalités, on a $x = y$, et s'il y a une égalité on a $x < y$). De même, si $a > c$ et $b < d$ on a $x > y$. Par ailleurs, si $a < b$ et $c > d$, on a $x < 1 < y$ d'où $x < y$, et l'on peut renverser le sens des inégalités. Dans le cas où aucun de ces critères ne donne de résultat, on compare ad et bc. On évite ainsi au maximum les multiplications, de plus, les tests à zéro (tests de signe) sont triviaux.

3.4 Addition et multiplication

Les deux fonctions qui additionnent ou multiplient des nombres sont des fonctions génériques de Lisp, à nombre arbitraire d'arguments. En règle générale, elles utilisent un résultat partiel, et ajoutent chaque nouvel élément à ce résultat partiel. On se ramène ainsi à une opération binaire.

Examinons d'abord les cas les plus simples : pour additionner (ou multiplier) deux nombres complexes $x = a + ib$ et $y = c + id$, on utilise les formules $x + y = (a + c) + i(b + d)$ et $xy = (ac - bd) + i(ad + bc)$, et l'arithmétique générique pour faire les calculs. Si un seul des deux nombres est complexe, par commutativité, on peut supposer que c'est x, et on applique les relations précédentes avec $c = y$ et $d = 0$. Si l'un des arguments est un flottant, l'autre argument est converti en flottant. Ce mécanisme est imposé par Lisp, et peut poser problème en cas de dépassement de capacité : il faudrait dans ce cas convertir l'argument en BigFloat. Ce problème n'est pas résolu à l'heure actuelle.

Nous considérons maintenant le cas de la somme ou produit de deux nombres, qui sont des chiffres, des entiers ou des fractions, en traitant d'abord la somme, puis le produit. On peut

évidemment supposer les deux nombres non nuls (0 est élément neutre pour la somme et absorbant pour le produit). Ainsi que cela a été fait pour la comparaison, il suffit de se ramener au cas suivants

- Somme de deux chiffres
- Somme d'un grand entier et d'un chiffre
- Somme de deux grands entiers
- Somme d'une fraction et d'un entier
- Somme de deux fractions.

Algorithme 13. (Addition des entiers et rationnels) *Les variables et paramètres utilisés ici sont x et y des nombres, a, b, c, d, N, D, p, X et Y des entiers internes,* is-simplified, is-not-integer, sign1 *et* sign2 *des variables globales booléennes.*

Procédure testint. *Paramètre x, un chiffre ou un entier* [Extrait le signe et la valeur absolue].

a. *Si x est un entier, positionner dans* sign2 *son signe, rendre sa valeur absolue.*

b. *Si $x < E/2$, mettre* sign2 *à vrai et rendre x,*

c. *Sinon, mettre* sign2 *à faux, et rendre $E - x$.*

Procédure testint1. *Cette procédure est la même que* `testint` *mais utilise* sign1 *au lieu de* sign2.

Procédure rqspll. *Paramètres X et Y* [Rend la valeur absolue de la somme de X et Y, les signes sont dans sign1 et sign2, le signe du résultat est dans sign1].

a. *Si* sign2 *et* sign1 *sont égaux, rendre* `nadd`(X, Y).

b. *Si* `ncmp`$(X, Y) \neq -1$, [cas $X \geq Y$] *rendre* `ndiff`(X, Y).

c. *Sinon remplacer* sign1 *par sa négation et rendre* `ndiff`(Y, X).

Somme de deux chiffres. *Cette procédure calcule la somme de deux chiffres x et y.*

a. *Si x et y ont des signes distincts, rendre $x + y$* [addition des chiffres via la fonction **add**].

b. *Si x et y sont positifs, appeler* `mknumpos` *sur* `nadd`(x, y).

c. *Sinon, appeler* `mknumneg` *sur* `nadd`$(E - x, E - y)$.

Somme de deux entiers. *Cette procédure calcule la somme d'un entier x et d'un nombre y, qui est un entier ou un chiffre.*

1. *Positionner dans la variable* sign1 *le signe de x, et remplacer x par sa valeur absolue.*

2. *Appeler* `testint` *sur y.*

3. *Rendre* `zx(rqspll`(x, y)).

Somme d'une fraction et d'un entier. *Cette procédure calcule la somme d'une fraction x et d'un entier ou chiffre y.* [Si $x = a/b$ le résultat est $(a + by)/b$. Si la fraction a/b est réduite, il en sera de même de la fraction $(a + by)/b$. Comme x n'est pas entier le résultat ne le sera pas non plus].

1. *Si on veut un résultat réduit, réduire la fraction x.*

2. *Positionner dans* sign1 *le signe de* x*, dans* is-simplified *l'indicateur qui dit si* x *est réduit, et mettre* is-not-integer *à vrai.*

2. *Remplacer* y *par* `testint`(y)*, appeler* a *et* b *les numérateur et dénominateur de* x*.*

3. *Calculer le numérateur* N *du résultat par la formule* $N =$ `rqspll`$(a,$ `ntimes`$(b, y))$*.*

4. *Rendre* `qx`(N, b)*.*

Somme de deux fractions non réduites. *Cette procédure est appelée avec deux fractions* $x = a/b$ *et* $y = c/d$*, elle rend la somme non réduite* $(ad + bc)/(bd)$*.*

1. *Positionner dans* sign1 *et* sign2 *les signes de* x *et* y*.*

2. *Calculer* a*,* b*,* c *et* d*. Positionner* is-simplified *et* is-not-integer *à faux.*

3.a. *Si* $b = d$*, calculer le numérateur* $N =$ `rqspll`(a, c) *et le dénominateur* $D = b$*.*

3.b. *Sinon, calculer le numérateur* $N =$ `rqspll`$($`ntimes`$(a, d),$ `ntimes`$(b, c))$ *et le dénominateur* $D =$ `ntimes`(b, d)*.*

4. *Rendre* `qx`(N, D)*.*

Somme de deux fractions réduites. *Cette procédure est appelée avec deux fractions réduites* $x = a/b$ *et* $y = c/d$*, elle rend la somme, sous forme de fraction réduite.*

1. *Positionner dans* sign1 *et* sign2 *les signes de* x *et* y*.*

2. *Calculer* a*,* b*,* c *et* d*.*

3.a. *Si* $b = d$*, positionner* is-simplified *et* is-not-integer *à faux, calculer le numérateur* $N =$ `rqspll`(a, c) *et rendre* `qx`(N, d)*.*

3.b. *Dans le cas contraire, faire ce qui suit* [Soit p le pgcd de b et d, $b = b'p$, $d = d'p$ et $n = \pm ad' + \pm b'c$. Alors $x + y = n/(b'd'p)$ et n est premier avec $b'd'$. Il suffit donc de réduire le quotient n/p. On note que $n \neq 0$].

4. *Positionner* is-simplified *à vrai et calculer le pgcd* p *de* b *et* d *via la fonction* `npgcd`*.*

5. *Si ce pgcd est 1, exécuter les étapes 7 et 9* [cette dernière se simplifie vu que $p = 1$].

6. *Diviser* b *et* d *par* p *au moyen de* `nquomod`*.*

7. *Calculer le numérateur* $N =$ `rqspll`$($`ntimes`$(a, d),$ `ntimes`$(b, c))$ *et le dénominateur* $D =$ `ntimes`(b, d)*.*

8. *Diviser* N *et* p *par leur pgcd.*

9. *Rendre* `qx`$(N,$ `ntimes`$(p, D))$*.*

Donnons maintenant l'algorithme du produit.

Algorithme 14. (Produit de deux entiers ou fractions non nuls.) *Les paramètres et variables locales utilisés sont* x *et* y *des nombres,* a*,* b*,* c*,* d*,* N*,* D*,* p*,* X *et* Y *des entiers internes,* is-simplified*,* is-not-integer*,* sign1 *et* sign2 *des variables globales booléennes.*

Procédure rqstimes1. *Cette procédure prend en argument une fraction* $x = a/b$ *et deux entiers internes* c *et* d*. Elle rend* cx/d *réduit.* [Elle suppose que x et c/d sont des fractions réduites, les signes sont précalculés].

1. *Calculer* a *et* b*, positionner* is-simplified *à vrai.*

2. *Calculer le pgcd* p *de* a *et* d*, s'il n'est pas trivial, diviser* a *et* d *par* p*.*

3. *Calculer le pgcd p de c et b, s'il n'est pas trivial, diviser c et b par p.*

4. *Calculer le numérateur $N =$* `ntimes`(a, c) *et le dénominateur $D =$* `ntimes`(b, d).

5. *Rendre* `qx`(N, D).

Produit de deux chiffres. *Les paramètres sont x et y, deux chiffres.*

a. *Si $x = 1$ rendre y et si $y = 1$ rendre x.*

b. *Si $x = -1$ rendre $-y$ et si $y = -1$ rendre $-x$.*

c.1. *Si $x < E/2$ et $y < E/2$, positionner* sign1 *à vrai, si $x > E/2$ et $y > E/2$, positionner* sign1 *à vrai, remplacer x par $E - x$ et y par $E - y$, sinon positionner* sign1 *à faux, et remplacer x (resp. y) par $E - x$ (resp. $E - y$) s'il est $> E/2$.*

c.2. *Rendre* `zx`(`ntimes`(x, y)).

Produit d'un entier par un chiffre ou un entier. *On suppose que cette procédure est appelée avec un grand entier x et un entier ou chiffre y.*

a. *Si $y = 1$ rendre x.*

b. *Si $y = -1$ rendre $-x$.*

c.1. *Remplacer x par* `testint1`(x) *et y par* `testint`(y).

c.2. *Si* sign2 *est faux, remplacer* sign1 *par sa négation.*

c.3. *Rendre* `zx`(`ntimes`(x, y)).

Produit d'une fraction par un entier. *Cette procédure est appelée avec une fraction x et un entier ou chiffre y.*

1. *Traiter les cas triviaux $y = 0$, $y = 1$ ou $y = -1$.*

2. *Remplacer y par* `testint`(y). *Si* sign2 *est égal au signe de x, positionner* sign1 *à vrai, à faux sinon.*

3.a. *Si y est égal au dénominateur de x, rendre le numérateur de x avec le signe* sign1.

3.b. *Si le résultat doit être réduit, commencer par réduire x, et rendre* `rqstimes1`$(x, y, 1)$.

3.c. *Sinon, positionner* is-simplified *et* is-not-integer *à faux, et rendre via* `qx` *la fraction dont le signe est* sign1*, le dénominateur est celui de x, et le numérateur est* `ntimes`(a, y) *où a est le numérateur de x.*

Produit de deux fractions. *Cette procédure rend le produit de deux fractions $x = a/b$ et $y = c/d$.*

1. *Calculer a, b, c et d, calculer dans* sign1 *le signe du résultat, positionner* is-simplified *et* is-not-integer *à faux.*

2.a. *Si $a = d$, rendre* `qx`(c, b).

2.b. *Si $b = c$, rendre* `qx`(a, d).

2.c. *Si le résultat n'a pas besoin d'être réduit, rendre $(ac)/(bd)$, le produit est calculé par* `ntimes`*, le quotient par* `qx`.

2.d. *Sinon, réduire d'abord x et y* [ceci change c et d] *et rendre* `rqstimes1`(x, c, d).

3.5 Division entière

On ne considère pas dans cette section la fonction générique `/`, qui divise le premier argument par le produit des autres en appelant les fonctions génériques du produit et d'inverse, expliquées plus haut, mais la fonction **quomod** qui rend le quotient entier.

Cette fonction positionne dans la variable globale `#:ex:mod` le reste de la division, et rend le quotient. Elle est utilisée en interne pour les deux fonctions **quotient** et **modulo**. C'est une fonction générique à deux arguments. Elle n'est cependant pas commutative, ce qui va poser certains problèmes. Par définition, le quotient q et le reste r de la division de a par b satisfont $a = bq + r$, $0 \leq r < |b|$, et q est un entier.

D'après cette définition on voit de suite que a et b doivent être des nombres réels. Nous avons choisi de plus l'option que a et b ne peuvent être des nombres flottants. Ce sont donc des entiers ou des fractions. Notons que si a et b sont positifs, le quotient est la partie entière de a/b, et que si a et b sont entiers, c'est la division usuelle. L'algorithme est le suivant (il est compliqué à cause de la gestion des signes).

Algorithme 15. (Division entière.) *Les variables et paramètres utilisés ici sont x et y des entiers, q, r des nombres. Toutes les procédures sont appelées avec x et y comme paramètres. Elles rendent le quotient, et positionnent le reste dans* `#:ex:mod`.

Procédure rqm2. *Procédure générique dans le cas de deux paramètres positifs x et y.*

1. *Calculer q la partie entière de x/y.*

2. *Calculer le reste $r = x - qy$.*

Procédure rquomod. *Procédure générique. Les deux arguments sont x et y. On sait que x est un chiffre, un entier ou un rationnel.*

1. *Tester que y est bien un chiffre, un entier ou un rationnel, et calculer son signe.*

2.a. *Si le signe est 0, c'est une erreur.*

2.b. *Si le signe est négatif, diviser x par $-y$ au moyen de* **quomod**. *Si le quotient de cette division est q, rendre $-q$, le reste étant inchangé.*

2.c. *Calculer le signe de x* [On suppose ici $y > 0$].

2.c.a. *Si le signe est 0* [i.e. $x = 0$] *le quotient et le reste sont 0.*

2.c.b. *Si le signe est positif, appeler* `rqm2`.

2.c.c. *Sinon, diviser $-x$ par y via la fonction générique* **quomod**. *Elle rend un quotient q' et un reste r'. Si $r' = 0$, rendre $-q'$ et r', sinon $-q' - 1$ et $y - r'$, en utilisant l'arithmétique générique pour faire ces calculs.*

Cas où le premier argument est un chiffre. *On suppose que x est un chiffre.*

a. *Si $x \geq 0$, $y > 0$ et y est un entier* [donc $0 \leq x < y$], *le quotient est 0 et le reste est x.*

b. *Appeler sinon la procédure* `rquomod` [Ceci suppose que si y est un chiffre, la fonction générique **quomod** de Lisp sait traiter le cas de deux chiffres, dans le cas contraire, il faudrait, dans le cas où y est un chiffre, faire le même traitement que dans le cas où x est un entier].

Cas où le premier argument est un entier. *On suppose que x est un entier.*

a. *Si $x > 0$ et y est un chiffre ou un entier, et $y > 0$, calculer le quotient via* `nquomod` *et le convertir en entier* [c'est un nombre interne]. *Convertir le reste, qui est dans* `#:ex:mod`, *en entier* [c'est aussi un nombre interne]. *Ces deux entiers sont positifs ou nuls.*

b. *Appeler sinon la procédure* `rquomod`.

Cas où le premier argument est un rationnel. *Appeler dans tous les cas la procédure générique* `rquomod`.

Chapitre 4

Arithmétique générique et complexe

Un nombre complexe est formé de deux champs, une partie réelle et une partie imaginaire, ce sont deux nombres réels. Contrairement à l'option choise par Common Lisp, les parties réelles et imaginaires peuvent être de type différent. Pour l'instant, ce sont des rationnels ou des flottants, on espère un jour autoriser des BigFloat, mais cela pose des problèmes de conversions, comme nous allons le voir dans la suite.

4.1 Types et conversions

La fonction **integerp** rend vrai si son argument est un entier ou un chiffre. Rappelons qu'une fraction n'est jamais un entier, donc que contrairement à la documentation Lisp, la fonction **integerp** appelée sur une fraction ne la réduit pas. La fonction **rationalp** rend vrai si son argument est un rationnel ou un entier (ou un chiffre). La fonction **realp** rend vrai si son argument est un réel (un flottant, un BigFloat, ou un objet pour lequel **rationalp** rend vrai). La fonction **complexp** rend vrai si son argument est un nombre complexe (ceci inclut les nombres réels, donc n'est pas consistant avec la définition de Common Lisp). La fonction **truecomplexp** rend vrai si son argument est un nombre dont la représentation interne est celle d'un nombre complexe (ceci exclut les nombres réels).

Les fonctions **realpart** et **imagpart** rendent la partie réelle et imaginaire d'un nombre complexe. Si ce nombre est réel, la première rend le nombre, la seconde rend 0 (ou 0.0 si le nombre est flottant).

La fonction **float** convertit son argument en flottant. Elle n'est pas définie pour les complexes, et peut provoquer des débordements si son argument est trop grand. Si l'argument est une fraction, elle est réduite au préalable (pour éviter dans certains des dépassements de capacité).

La fonction **fix** ou **floor** rend la partie entière (au sens usuel) d'un nombre. Elle n'est définie que pour les nombres réels. Il peut y avoir des problèmes de précision pour les nombres flottants. La fonction **ceiling** convertit un nombre en entier en tronquant dans l'autre sens, la fonction **truncate** convertit son argument en tronquant en direction de 0. La fonction **round** est définie comme en Lisp. Sa sémantique est peu claire, elle prend deux arguments. Pour plus de détails, on

peut consulter [4, p. 216], en remarquant toutefois que nos fonctions ne rendent qu'une valeur, et que même si la variable `#:ex:mod` contient parfois le reste correct, ce n'est qu'un heureux hasard.

4.2 Fonctions trigonométriques

La fonction **phase** appliquée à un nombre complexe c rend sa phase. Si le module (valeur absolue) est r, la phase ϕ est telle que $c = re^{i\phi}$. Cette phase est entre $-\pi$ et π, elle est calculée en utilisant **atan2** qui est une fonction à deux arguments x et y qui calcule **atan** (l'arc tangente) du quotient x/y, mais utilise les signes de façon conventionnelle (voir par exemple [4, p. 208]) pour avoir une détermination du résultat entre $-\pi$ et π. La fonction **conjugate** rend le conjugué de son argument si c'est un complexe, son argument sinon.

Les fonctions **sin**, **cos**, **tan**, **asin**, **acos**, **atan**, leur équivalent hyperbolique, de même que les fonctions **log**, **log10**, **sqrt**, **power** et **exp** existent. Elles font leurs calculs en complexe ou flottant. La méthode de calcul n'est pas optimale. Pour l'instant il n'y a pas moyen de faire des calculs en BigFloat (par exemple, si on appelle **asin** sur l'entier 1, le résultat est $\pi/2$, et l'arithmétique générique n'est pas appelée).

Notons que la fonction ****** prend deux arguments x et n et calcule x^n. Dans le cas où n n'est pas entier, ceci se fait en appelant la fonction **power**, mais sinon elle utilise des multiplications.

Notons également que la fonction **powermod** prend trois arguments a, b et c. Elle calcule a^b modulo c où $c > 0$ et $b \geq 0$ de façon efficace. L'inverse de a modulo c peut être calculée via la fonction **bezout**.

4.3 Autres fonctions

La fonction **fib** calcule un nombre de Fibonacci F_n, où F_n, défini par $F_1 = F_2 = 1$ et $F_n = F_{n-1} + F_{n-2}$, vérifie la propriété $F_{2n} = F_n(2F_{n+1} - F_n)$, $F_{2n+1} = F_{n+1}^2 + F_n^2$ et $F_{2n+2} = F_{n+1}(2F_n + F_{n+1})$. Pour $n \leq 22$ on utilise la définition itérative, et sinon, les trois formules précédentes. Pour être plus efficace, les fonctions génériques sont remplacées par des appels à `ntimes` et `nadd`. A titre d'exemple, le calcul de F_{10000} prend 0.48 secondes sur une Sparc Station 2, le résultat est un nombre avec environ 2000 chiffres, et le nombre s'imprime en 1.2 secondes.

La fonction *(**fact** n)* calcule la factorielle de n pour l'entier n ($n < 32767$) via $fact(1, 1, n)$. La fonction $fact(g, i, d)$ calcule le produit $g(g + i) \ldots (g + \alpha i)$ où α est le plus grand entier tel que $g + \alpha i \leq d$. Si le nombre de termes dans le produit est au plus 10 (i.e. $d - g < 10i$), on fait les produits, sinon on calcule le produit de $fact\ (g, 2i, d)$ et $fact(g + i, 2i, d)$. Notons que l'on a $g \leq i \leq 4096$ et $g \leq d$. Par conséquent le seul cas d'overflow qu'on peut avoir sur la manipulation des chiffres g, i, d et k est le produit de 10 par i. Cette astuce permet d'éviter dans la plupart des cas l'arithmétique générique sur les indices. Pour les produits de grands entiers, on utilise `ntimes`.

A titre d'exemple, la factorielle de 10000 se calcule en 115 secondes, le résultat a de l'ordre de 35 000 chiffres, le temps d'impression est de 336 secondes. A titre de comparaison, le logiciel de calcul formel Maple (qui possède une arithmétique en base 10) calcule ce nombre en 230 secondes, et l'imprime en 20 secondes (bien entendu, comme la base est 10, l'algorithme d'impression est trivial, il faut cependant du temps pour imprimer les 35 000 chiffres, en moyenne une seconde par page d'écran).

Les fonctions **gcd** et **bezout** calculent le pgcd et la relation de Bezout entre deux entiers. Nous avons expliqué le code interne `npgcd` et `nbezout` de ces fonctions. Les fonctions utilisateur n'ont donc qu'à s'occuper des signes des quantités.

En ce qui concerne la relation de Bezout, la fonction **bezout** appliquée à deux entiers x et y rend une liste de trois entiers, le pgcd p et deux entiers tels que $ux + vy = p$. On sait que la procédure interne `nbezout` rend des nombres non signés, u et p. En premier lieu on transforme p en nombre positif, u en nombre signé et on calcule v par cette relation. Par ailleurs cet algorithme suppose x et y positifs. Pour ce faire, on l'appelle avec les valeurs absolues de x et y. Le cas où l'un des deux arguments x et y est 0 est trivial. Dans le cas où les deux arguments sont nuls, par convention, on rend 0 pour les trois quantités. Finalement, la fonction interne suppose $x > y$. Si $x < y$ il suffit d'échanger x et y, puis u et v à la fin du calcul. Et si $x = y$, le pgcd est x, et on a le choix entre $u = 0, v = 1$ ou $u = 1, v = 0$. Ceci est le seul cas où les quantités u, v et p ne sont pas uniques ous les hypothès $p > 0$, $|u| < |y|/p$ et $|v| < |y|/p$.

4.4 Fonctions logiques

Nous avons étendues les fonctions logiques à des nombres de taille arbitraire. Pour ce faire, on considère que les nombres négatifs sont représentés en complément à 2.

Un nombre positif doit être considéré comme une suite infinie à gauche de bits, dont seul un nombre fini sont distincts de 0, donc $\ldots 0 \ldots 0 x_n \ldots x_0$ qui correspond au nombre $\sum_{i=0}^{n} 2^i x_i$. Le complément à 1 d'un nombre est obtenu en échangeant les bits 0 et 1. Le complément à 2 est alors obtenu en additionnant 1 au résultat. Par conséquent, si le nombre se termine par k bits nuls, on laisse les $k+1$ derniers bits inchangés, et on échange les 0 et les 1 pour les autres bits. Un nombre négatif est donc une suite infinie de bits dont seul un nombre fini sont différents de 1. La fonction `compl2` calcule le complément à 2 (tronqué sur n mots) d'un nombre en travaillant sur les mots de la façon suivante : en cherche le premier mot non nul en partant de la droite. On prend son complément à 2 en prenant son opposé (la représentation interne de Lisp est une représentation en complément à deux). Pour les autres mots on prend leur complément à 1 au moyen de la fonction **lognot**.

La fonction **lsh** prend deux arguments x et y. Elle décale x de y positions. Cette fonction est simple : elle rend un objet qui a même signe que x et regarde la valeur absolue. Dans le cas où $y > 0$, le résultat est $x2^y$, sinon c'est la partie entière du quotient de x par 2^{-y}. On utilise une procédure auxilliaire qui écrit y sous la forme $a2^{16} + b$, où $a < 2^{15}$, car sinon, le résultat est trop gros (sauf si $y < 0$, auquel cas on rend 0 de suite). Il s'agit de décaler x et a mots, et de b bits. Le décalage en mots se fait par recopie du vecteur dans un vecteur plus grand ou plus petit, le décalage en bits se fait en utilisant `ntimes` (cas d'une multiplication) ou `nquomod` (cas d'une division).

Les opérations **land**, **lor** et **lxor** calculent le *et* logique, *ou* logique ou le *ou exclusif*. Elles utilisent des fonctions internes qui supposent que leur deux arguments sont des vecteurs de même taille, et utilisent les primitives Lisp **logand**, **logor** et **logxor** sur chaque mot. Nous ne décrirons pas ici ces trois fonctions, uniquement la fonction **land** (le principe étant le même pour ces trois fonctions).

Soit à calculer le *et* logique entre deux entiers x et y. Dans un premier cas les deux nombres sont positifs. Le résultat est alors un nombre positif, dont la taille est au plus la taille du plus petit des deux nombres. Il suffit donc de copier le plus grand, et d'appeler la fonction interne sur cette copie. Il faudra éventuellement supprimer les zéros en tête. Notons que si l'un des deux

arguments est un chiffre, on n'a pas besoin de copier quoi que ce soit. Dans un deuxième cas, l'un des arguments est négatif (par symétrie, on peut supposer que c'est y) et l'autre est positif. Il faut prendre le complément à 2 de y, et le résultat est positif. Finalement si les deux nombres sont négatifs, on prend le complément à deux de chaque nombre, et on réalise l'opération interne, et comme le résultat est négatif, on prend le complément à 2 du résultat. En règle générale, on copie les deux vecteurs dans des vecteurs qui ont comme taille la plus grande des deux tailles, avant de calculer les compléments à 2.

Finalement, la fonction **haulong** donne le nombre de bits de la valeur absolue d'un entier.

4.5 Entrées-sorties

La représentation externe d'un entier est la suite de ses caractères. Le programme d'impression utilise la variable-fonction **obase** pour déterminer la base dans laquelle imprimer les chiffres. Si la base est plus grande que 10, on utilise des lettres majuscules à la place de chiffres, en commençant par la lettre A. Le programme interne d'impression des entiers a été décrit au premier chapitre. La fonction de lecture utilise la variable-fonction **ibase** pour trouver la base. Pour entrer un entier, il suffit de donner la suite de ses chiffres. Notons qu'il ne faut pas donner plus de chiffres que ne peut contenir le tampon interne de lecture de Lisp. Ce tampon est de taille 256 (d'après la documentation) mais suivant les implémentations peut être plus grand (400 par exemple). Il n'y a pas de limitations si la lecture ne se fait pas sur le tampon interne, typiquement pour les deux fonctions **implode** et **stratom** (cette dernière fonction est utilisée en interne par le lecteur Sisyphe, il n'y a donc pas de limitation de taille sur les entiers dans Sisyphe).

Il est possible de préciser la base de lecture au moyen d'une construction de la forme `#12R25` (25 en base 12). Dans ce cas le 12 est lu en base 10. Notons que la documentation Lisp précise que `#12r25` rend la même résultat. La différence essentielle est que, dans le cas de `#12r25`, si la base n'est pas 10, les calculs sont faits modulo E (donc 2^{16}). Une autre différence est que le macro caractère R ignore les espaces et les changements de lignes après un backslash. En d'autres termes, le nombre peut être fourni sur plusieurs lignes, il n'y a pas de limite sur la taille du nombre.

Les nombres rationnels sont lus et imprimés sous la forme `1/2`. Il peuvent bien entendu être donnés sous la forme `#12R25/3`.

Il est possible d'imprimer des nombres rationnels de façon différente, en utilisant la fonction **prin-rational**. Elle prend trois arguments, un rationnel x, un type T, et une précision p. Si le type est *float0*, un développement décimal, avec au plus p chiffres sera imprimé, et la période est imprimée avec des accolades. Si le type est *float1*, un développement décimal, avec au plus p chiffres sera imprimé. La période ne sera pas imprimée dans ce cas. Si le type est *cf0*, *cf1* ou *cf-1*, le développement en fraction continue avec au plus p termes sera imprimé. Dans ce dernier cas, si p est négatif, le développement en fraction continue est retourné au lieu d'être imprimé. La fonction **cf-to-rational** au contraire transforme une fraction continue en nombre rationnel.

Les nombres complexes peuvent être lus et imprimés sous la forme `#C(1 2)`. Pour rendre cette notation plus lisible, on peut changer une variable globale de sorte que le résultat imprimé soit `[2i+1]`.

Finalement, pour pouvoir relire rapidement des nombres, si la variable **#:system:print-for-fast-read** et vraie, et si **#:system:print-for-read** est également vraie, les entiers et les fractions sont imprimés dans leur représentation interne, précédée de `#10R/1`. Ici le 1 signifie que c'est la première version du logiciel. On espère pouvoir relire en LeLisp version 16 les nombres imprimés par ce logiciel.

Bibliographie

[1] J. Hervé, F. Morain, D. Salesin, B. Serpette, J. Vuillemin, and P. Zimmermann, *BigNum: Un module portable et efficace pour une arithmétique à précision arbitraire*, Rapport de Recherche 1016, INRIA, 1989.

[2] INRIA, *Le_Lisp de l'INRIA Version 15.22, Le Manuel de Référence*, INRIA ed., Janvier 1989.

[3] D. Knuth, *The Art of Computer Programming*, Addison Wesley, 1981.

[4] G. L. Steele Jr, *Common LISP: The Language*, Digital Press, Burlington, MA., 1984.

[5] J. Vuillemin, *Exact real computer arithmetic with continued fractions*, Rapport de Recherche 760, INRIA, Nov. 1987.

Table des matières